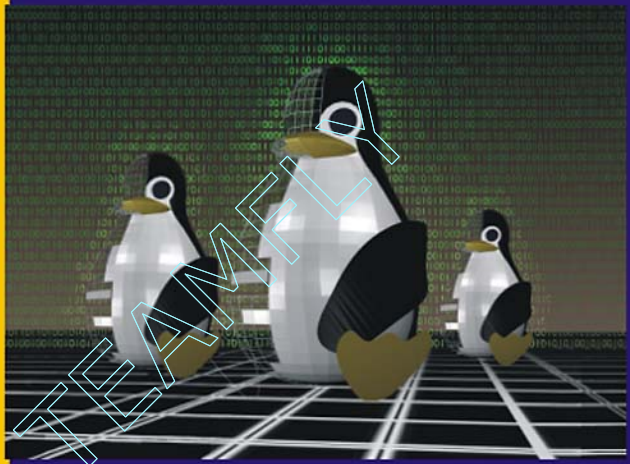
WORDWARE GAME DEVELOPER'S LIBRARY

Learn the 3D graphics techniques used in advanced applications, such as portals, 3D BSP trees, light mapping, shadows, particle systems, and more, all with working C++ code.

Understand sophisticated 3D modeling techniques using Blender, including inverse kinematics, rotoscoping, and development of a complete world editing system.

Create complete interactive 3D environments or games using collision detection, TCP/IP networking, physics, digital sound, and advanced 3D content development systems such as the World Foundry GDK.



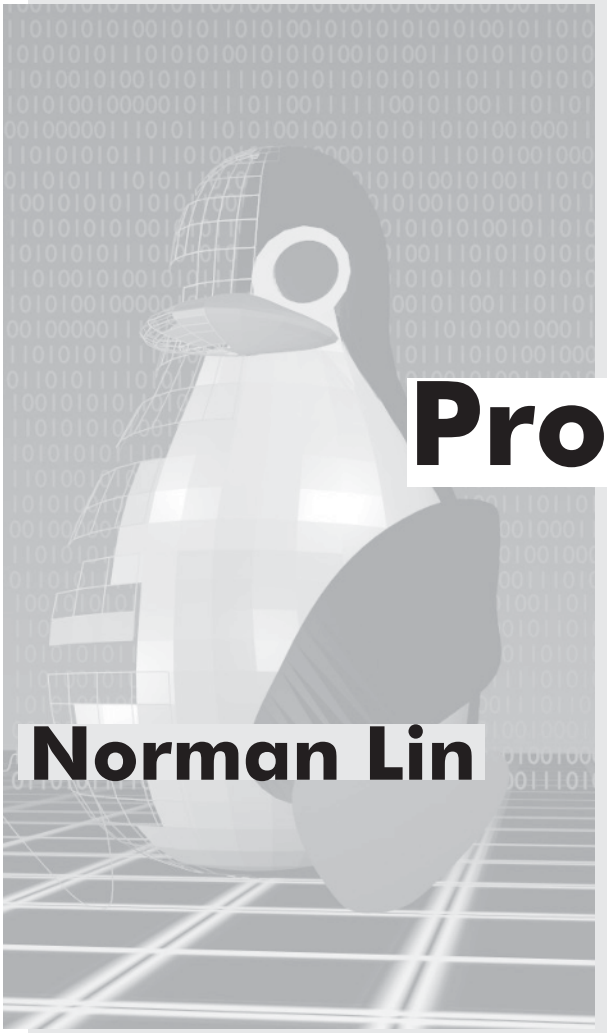
TEAMFLY

Advanced Linux 3D Graphics Programming

Norman Lin

Companion CD-ROM Included

WORDWARE Publishing Inc.



Advanced Linux 3D Graphics Programming

Norman Lin

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Lin, Norman.
Advanced Linux 3D graphics programming / by Norman Lin.
p. cm.
Includes index.
ISBN 1-55622-853-8 (pbk.)
1. Computer graphics. 2. Linux. 3. Three-dimensional display systems. I. Title.

T385 .L5555 2001
006.6'93--dc21 2001026370
CIP

© 2001, Wordware Publishing, Inc.

All Rights Reserved

2320 Los Rios Boulevard
Plano, Texas 75074

No part of this book may be reproduced in any form or by
any means without permission in writing from
Wordware Publishing, Inc.

Printed in the United States of America

ISBN 1-55622-853-8
10 9 8 7 6 5 4 3 2 1
0106

Blender is a registered trademark of Not a Number B. V.
Other product names mentioned are used for identification purposes only and may be trademarks of their respective companies.

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

Contents

Acknowledgments x
Preface xi
Introduction xii

Chapter 1 **Basic Linux 3D Graphics Concepts 1**
2D Graphics Fundamentals 1
3D Graphics Fundamentals 3
3D Coordinate Systems and Vectors 4
Perspective Projection 5
Matrices 6
Specific Matrix Transformations 6
Other Matrix Properties 7
The l3d Library Classes 8
Sample l3d Program 8
l3d Directory Structure 12
The Five-Step Process of l3d Programs 13
Overview of l3d Classes 19
Applications and Events 19
2D Graphics 20
Concrete Factory Management 24
Specifying Geometry and Behavior 25
Fixed- and Floating-Point Math 29
Summary of l3d Classes 31
Linux Programming Tools 32
Linux 3D Modeling 32
Blender Interface and Commands 33
Exporting and Importing Blender Models 36
Summary 37

Chapter 2 **Rendering and Animation Techniques for 3D Polygons 39**
Vertex Animation and 3D Morphing 39
Sample Program: morph3d 40
Lighting 48
Mathematical Models for Computing Light 49
Self Lighting 49
Ambient Lighting 50
Diffuse Reflection 50
Specular Reflection 55
Multiple Light Sources and Components 57
Radiosity and Ray Tracing 58
Dynamic or Static Lighting Computations 58
Fog 59
Rendering Techniques for Drawing Light 60
Flat Shading 61

Gouraud Shading . . . 61
Phong Shading . . . 63
Light Maps . . . 63
Texture Mapping . . . 64
Step 1: Define a Texture . . . 65
Storing Texture Data . . . 66
Classes for Loading Textures from Disk . . . 68
Practical Issues in Dealing with Texture Image Files . . . 73
Step 2: Define a Texture Space . . . 74
Step 3: Map Between Texture Space and World Space . . . 76
Calc: A Symbolic Algebra Package . . . 79
Starting and Exiting Calc . . . 79
Stack-Based Computation . . . 80
Entering and Editing Mathematical Entities . . . 82
Solving Systems of Equations . . . 85
Solving the Texture Mapping Equations with Calc . . . 86
Step 4: Reverse Project from Screen Coordinates into Texture Coordinates . . . 89
Step 5: Map Texture Coordinates to Integer Indices and Draw . . . 92
An Optimized Texture Mapping Strategy: u/z, v/z, 1/z . . . 93
The Division Operation and Texture Mapping . . . 95
Associating Textures with 3D Polygons . . . 96
Rasterization Classes for 3D Polygons . . . 98
An Abstract 3D Rasterizer: l3d_rasterizer_3d . . . 98
A Software 3D Rasterizer Implementation: l3d_rasterizer_3d_sw_imp . . . 101
A Mesa/OpenGL 3D Rasterizer Implementation: l3d_rasterizer_3d_mesa_imp . . . 115
Sample Program: textest . . . 129
Light Mapping Revisited . . . 135
Software Light Mapping . . . 136
Surfaces . . . 136
Surface Cache . . . 141
Light Mapped Polygons . . . 142
Software Rasterization of Light Maps . . . 147
Hardware Light Mapping . . . 147
Sample Program: lightmap . . . 151
Shadows and Light Maps . . . 159
Summary . . . 160

Chapter 3 **3D Modeling with Blender . . . 161**
Tutorial: Creating and Exporting Compatible, Textured 3D Morph Targets . . . 161
The Starting Morph Mesh . . . 162
Inserting Two Morph Targets into Blender . . . 163
Deforming the Mesh . . . 165
Applying a Texture and Assigning Texture Coordinates . . . 167
Testing the Morph in Blender . . . 173
Exporting the Two Morph Targets . . . 173
Exporting the Texture Information . . . 174
Importing the Morph Targets into a Program . . . 175
Tutorial: Using Inverse Kinematics and Roto- scoping to Model a Walking Human Figure . . . 180
Inverse Kinematics: Definition . . . 181
Creating an Ika Chain in Blender . . . 183

Working with Ika Chains . . . 183
Creating the Arm Ikas . . . 184
Creating the Main Body Ika . . . 185
Parenting the Ikas into a Hierarchy . . . 185
Testing the Ika Chains . . . 187
Animating the Ika Chains . . . 188
Connecting Ika Chains and Meshes . . . 189
Texturing and Exporting the Model . . . 190
Importing the Textured Ika Meshes . . . 192
Rotoscoping and Inverse Kinematics . . . 197
Programming IK and FK . . . 200
Summary . . . 201

Chapter 4 **Visible Surface Determination I: General Techniques . . . 203**
The Goals of VSD . . . 204
Back-Face Culling . . . 207
3D Convexity and Back-Face Culling . . . 209
Sample Program: backface . . . 209
Class l3d_World_Backface . . . 214
View Frustum Culling . . . 218
Defining a View Frustum . . . 218
Computing the Frustum in World Coordinates . . . 220
Class l3d_Viewing_Frustum . . . 221
Using the Frustum Planes . . . 223
Hierarchical View Frustum Culling . . . 223
Bounding Spheres and the View Frustum . . . 225
Computing Bounding Spheres . . . 227
Class l3d_bounding_sphere . . . 228
Other Bounding Volumes . . . 231
Clipping Against the View Frustum . . . 233
Sample Program: frustum . . . 233
Class l3d_World_Frustum . . . 236
The Painter's Algorithm . . . 242
The Z Buffer Algorithm . . . 245
General Observations about the Z Buffer . . . 246
A Software Z Buffer: Class l3d_rasterizer_3d_zbuf_sw_imp . . . 248
Mesa/OpenGL Z Buffering . . . 257
Factory Manager for Z Buffered Rasterizers . . . 261
Sample Program: texzbuf . . . 263
Z Buffer-like Algorithms . . . 264
Summary . . . 266

Chapter 5 **Visible Surface Determination II: Space-partitioning Techniques . . . 267**
Binary Space Partitioning Trees, Octrees, and Regular Spatial Partitioning . . . 267
Using a BSP Tree to Partially Pre-sort Polygons . . . 271
Choosing a Splitting Plane . . . 272
Back-to-Front Rendering (Painter's Algorithm Revisited) . . . 274
Front-to-Back Rendering . . . 275
Combining BSP Trees and Bounding Volumes . . . 275
Sample Program: bsp . . . 276

Classes l3d_halfspace and l3d_bsptree . . . 277
Class l3d_world_bsptree . . . 286
The Main Program . . . 290
The World Database, Revisited . . . 293
Leafy BSP Trees: Automatic Convex Partitioning of Space . . . 293
Creating a Leafy BSP Tree . . . 295
Methods for Leafy BSP Trees in Class l3d_bsptree . . . 296
Sample Program: leafybsp . . . 297
Axis-aligned BSP Trees and Mini BSP Trees . . . 302
BSP Tree as a Multi-resolution Solid-Modeling Representation . . . 303
BSP Trees and Dimension Independence . . . 306
Octrees . . . 306
Regular Spatial Partitioning . . . 308
Portals and Cells . . . 308
The Main Ideas Behind the Portal Algorithm . . . 308
Rendering a Portal World . . . 310
Observations about the Portal Scheme . . . 313
Portals as a Connectivity Graph . . . 313
Advantages and Disadvantages . . . 313
Back-Face Culling . . . 314
Clipping . . . 314
Convexity or Non-Convexity . . . 315
Moving the Camera and Objects Within a Portal Environment . . . 315
Portals and the Near Z Plane . . . 316
Shadows . . . 318
Mirrors . . . 320
Portals and Other Rendering Methods . . . 321
Classes for Portals and Sectors . . . 322
Class l3d_polygon_3d_portal . . . 322
Class l3d_sector . . . 323
Class l3d_world_portal_textured_lightmapped_obj . . . 329
Class l3d_rasterizer_2d_sw_lighter_imp . . . 344
Class l3d_pipeline_world_lightmapped . . . 351
Sample Program: porlotex . . . 353
Other VSD Algorithms . . . 356
Summary . . . 357

Chapter 6 **Blender and World Editing . . . 359**
World Editing . . . 360
No World Editor . . . 360
Write Your Own World Editor . . . 361
Adapt an Existing Editor . . . 362
Using Blender for Portal Worlds . . . 363
Main Ideas of a Blender Portal World Editor . . . 364
Step-by-Step Guide to World Design . . . 367
Data Flow within the World Editing System . . . 368
Creating Sectors and Portals . . . 369
Tutorial: Creating Aligned Portals via Extrusion and Separation . . . 371
Tutorial: Aligning Portals from Separate Meshes . . . 374
Tips for Working with Portals . . . 382
Portalization: Generating Portal Connectivity . . . 385

Perl Scripts . . . 387
Architecture of the Perl Portalization System . . . 389
Structural Modules . . . 390
Parsing and Generator Modules . . . 415
Controlling Scripts . . . 429
Embedding Location, Orientation, Texture, Actor, and Other Information into Meshes . . . 430
Basic Ideas of Associating Attributes with Objects . . . 431
Store an ID, Location, and Orientation in Overlapping Edges . . . 431
The Tool Blend_at: Remote Control of Blender . . . 433
Configuration and Testing of blend_at . . . 434
Specific Mesh Attributes Used by the Portalization System . . . 437
The Name Attribute . . . 437
The Type Attribute . . . 437
Attributes for Sectors . . . 437
Attributes for Actors . . . 439
Parsing of Attributes by VidscParser.pm and vidinfo . . . 440
Program Listings for blend_at . . . 446
Class vertex . . . 447
Class blender_config . . . 447
Class blender_controller . . . 448
Class blender_xcontroller . . . 449
Tutorial: Creating a Textured Room with Actors . . . 463
Tips for Working with Attributes . . . 473
Summary of Blender and Portal Worlds . . . 474
Other World Editing Ideas . . . 475
Portalized Regular Spatial Partitioning . . . 475
BSP Tree and Octree . . . 476
Non-convex Sector-based Partitioning . . . 476
Summary . . . 478

Chapter 7 **Additional Graphics Techniques . . . 479**
Special Effects . . . 479
Environment Mapping . . . 480
Billboards . . . 484
Lens Flare . . . 486
Particle Systems . . . 487
Physics and Particle Systems . . . 488
Real-Time Update . . . 489
Sample Program: particle . . . 490
Comments on the Sample Program's Physics . . . 496
Some Ideas for You to Try . . . 496
Natural Phenomena . . . 497
Bump Mapping . . . 499
Multi-pass Techniques . . . 500
Advanced Techniques . . . 501
Curved Surfaces . . . 501
Level of Detail . . . 505
Billboards . . . 506
Edge Collapse . . . 506
BSP Tree . . . 507
Texture LOD Techniques: MIP Mapping . . . 508

Landscapes . . . 509
Storing Landscapes as Height Fields . . . 509
Generating Fractal Landscapes . . . 510
Rendering and LOD Techniques for Landscapes . . . 511
Camera Tracking . . . 512
Summary . . . 513

Chapter 8 **Non-Graphical Techniques for Games and Interactive Environments . . . 515**
Sound . . . 515
Basics of Digital Sound . . . 516
The RPlay Server . . . 519
Using TCP/IP Networking to Communicate with the Server . . . 520
Class l3d_sound_client . . . 521
Class l3d_sound_server_rplay . . . 522
TCP/IP Networking . . . 524
The Client . . . 524
The Server . . . 526
Running the Sample Server and Client . . . 529
Non-Blocking Operations . . . 529
What Data to Send . . . 530
Collision Detection . . . 530
Intersection Testing and Bounding Volumes . . . 531
Sphere-to-Sphere . . . 532
Ray-to-Polygon . . . 532
Ray-to-Sphere . . . 535
Sphere-to-Polygon . . . 536
Tunneling and Sweep Tests . . . 538
Multiple Simultaneous Collisions and Collision Response . . . 541
Allowing Penetration . . . 541
Avoiding Penetration with Temporal Search . . . 542
Class l3d_collidable . . . 543
Class l3d_collidable_sphere . . . 544
Class l3d_polygon_3d_collidable . . . 548
Class l3d_polygon_3d_textured_lightmapped_collidable . . . 551
Class l3d_camera_collidable . . . 552
Class l3d_world_portal_textured_lightmapped_obj_colldet . . . 553
Plug-in Object Seeker, Class l3d_plugin_videoscape_mesh_seeker . . . 563
Sample Program: collide . . . 574
More Advanced Collision Detection and Response . . . 576
Physics . . . 577
Some Basic Concepts . . . 577
Rigid Body Dynamics . . . 578
Real-Time Update and Numerical Integration . . . 579
Artificial Intelligence . . . 580
Summary . . . 582

Chapter 9 **What Lies Ahead? . . . 583**
Content Development Systems . . . 583
Game Blender/Blender 2.0 . . . 583
World Foundry . . . 590

What Does This Mean for 3D Programmers? . . . 598
The Future . . . 599
Summary . . . 600
Perspective . . . 600

Appendix . . . **603**
CD Installation . . . 603
License . . . 603
Contents of the CD-ROM . . . 603
Quick Start Guide . . . 604
Directories . . . 604
Installing the Sample Programs and Other Software . . . 605
Troubleshooting the Sample Programs . . . 607
Some Comments on the Sample Programs . . . 607
Hardware Acceleration . . . 608
Porting the Code to Microsoft Windows . . . 609
Tools Used to Prepare this Book . . . 610
Resources . . . 611
3D Graphics Programming . . . 612
3D Modeling . . . 612
3D Information and Applications . . . 613
General Programming . . . 613
Other . . . 614
References . . . 614

Index . . . **617**

Acknowledgments

In addition to my parents, Forest and Vicki Lin, I would like to thank the following individuals who directly or indirectly played a role in the completion of this book. Thanks go to my brother Tony, who persuaded me to download and try out the game *Doom*—an experience that convinced me that interactive 3D graphics on the PC was finally possible. Special thanks also to Stan Hall, who provided encouragement and advice even when it seemed that the book might not see the light of day.

Solveig Haring and Margit Franz were kind enough to provide me with Internet access and a cup of coffee for some of the longer nights in the computer lab. Ton Roosendaal provided some very interesting insights into Blender and 3D graphics in general. My work colleagues Horst Hörtner, Werner Pankart, Klaus Starl, and Thomas Wieser were all supportive and understanding during those times when work on the book required absence from the office. Andreas Jalsovec and Dietmar Offenhuber gave me insight into some of the nuances of 3D modeling. Renate Eckmayr, Viju John, Azita Ghassemi, Manfred Grassegger, Ulrike Gratzer, Andrea Groisböck, Jogi and Reni Hofmueller, Angelika Kehrer, Astrid Kirchner, Dietmar Lampert, Christine Maitz, Paula McCaslin, Bernd Oswald, Gabi Raming, Regina Webhofer, and other individuals too numerous to mention all expressed interest upon hearing that I was writing this book, and gave me much needed inspiration and motivation.

Professor Deborah Trytten got me started on the right track in 3D graphics during my studies at the University of Oklahoma. Kevin Seghetti carefully read and checked the text for technical accuracy and provided many valuable suggestions. Thanks also to everyone at Wordware Publishing, especially Jim Hill, who shared my enthusiasm about the book and was key in actually getting this project out the door.

Last but not least, I would like to thank the countless individuals around the world involved with the creation and maintenance of the freely available, high quality, open source GNU/Linux operating system and tools.

TEAM-FLY

Team-Fly®

Preface

A university professor of mine once mentioned that there was no danger that the computer science community would ever run out of interesting problems to solve. As a community, computer scientists try to understand the nature of computation by forming theories and attempting to prove their validity. We try to answer questions. Those theories which correctly capture the nature of computing problems contribute to the common pool of academic knowledge. Previously unanswered questions receive answers—some more complete, some less complete. The less complete answers raise new questions for further research; the more complete answers are eventually adopted by industry practitioners.

3D graphics is a field that illustrates this phenomenon well. In the early days, 3D graphics was mostly confined to academic research labs. The mathematics and geometry of 3D graphics were questioned and explored, and the field grew as a result. Today, research in 3D graphics is still very active, but at the same time, 3D graphics has also become mainstream. A number of graphics techniques from academia have established themselves as efficient and effective enough for widespread use. A 3D programmer should be familiar with these techniques. The purpose of this book is to communicate these important 3D techniques to the intermediate 3D programmer in a clear and intuitive way, using geometrical explanations supported with numerous working code examples.

This book uses Linux as the implementation platform. The free operating system has a number of advantages which make it ideal for learning and programming 3D graphics. The most important advantage is accessibility: the free, open source nature of Linux makes it possible for any programmer to have access to a top-quality operating system and development environment. This open nature has encouraged the development of massive amounts of free software (where free refers not only to cost, but mainly to the *freedom* to study and modify the source code), including software important for 3D graphics. Therefore, Linux offers any programmer the chance to get involved with 3D graphics programming today, at no cost, without forcing the programmer to either pay thousands of dollars in software licensing fees or to spend literally man-years of software development time creating customized tools. Linux already offers the tools you need to do serious 3D programming—and the freedom to use, learn from, and modify these tools.

This book builds upon the foundation laid in the introductory companion volume *Linux 3D Graphics Programming*. It is assumed that you have an understanding of all of the material presented in the introductory volume; the first chapter provides a quick review of this material. Therefore, this book is not suited for the complete beginner to 3D graphics. Such readers should work through the introductory companion book before attempting to read this book.

Introduction

Welcome, reader! I am glad to have you along and hope that you are as excited as I am about Linux and interactive 3D graphics programming. Take your time and enjoy the following few pages as we leisurely discuss the goals and contents of this book.

This book is the second volume of a two-volume work on interactive 3D graphics programming under Linux. First, let's look at the two-volume work as a whole, then we'll look more specifically at the contents of this volume.

Taken as a whole, the two-volume work aims to provide you with the knowledge, code, and tools to program top-notch, object-oriented, real-time 3D games and interactive graphics applications for Linux, which can also easily be ported to other platforms. By working through both volumes, you will learn to use the most important techniques, tools, and libraries for Linux 3D graphics: portals, OpenGL/Mesa, Xlib, 3D hardware acceleration, collision detection, shadows, object-oriented techniques, and more. We also cover the often neglected topic of 3D modeling, illustrating in detail how to use the professional 3D modeling package Blender, which is included on the CD-ROM, to create animated 3D models and portal worlds for use in our interactive 3D programs.

This second volume, titled *Advanced Linux 3D Graphics Programming*, covers more advanced techniques needed for realistic display of larger datasets often used in interactive 3D environments. Topics covered include: rendering and animation techniques for 3D polygons (3D morphing, texture mapping, light mapping, fog), the creation of more sophisticated 3D models with Blender (including jointed figures animated with inverse kinematics), importing such models from Blender into our programs, hidden surface removal (portals, BSP trees, octrees, *z* buffers), non-graphical issues relevant to interactive environments (special effects, collision detection, digital sound, TCP/IP networking, particle systems), and tutorials on using advanced 3D content development systems under Linux (Game Blender and World Foundry). Sample programs are provided, both in text form and on the CD-ROM, illustrating the concepts.

This book builds on the foundation laid by the introductory companion volume, *Linux 3D Graphics Programming*. The first chapter of this book serves as a brief review of the earlier material.

Goals of This Text

This text has several objectives. A primary goal of this text is to give you a solid understanding of the fundamental concepts involved in interactive 3D graphics programming at the intermediate to advanced level. Such an understanding not only enables you to write your own 3D programs, libraries, and games under Linux, but also gives you the knowledge and confidence you need to

analyze and use other 3D graphics texts and programs. In the open source world of Linux, understanding fundamental concepts is indeed important so that you can understand and possibly contribute to the common pool of knowledge and code. Furthermore, learning fundamental 3D graphics concepts also enables you to understand and effectively use sophisticated 3D applications and libraries such as 3D modelers and OpenGL.

A second goal of this text is to give you plenty of hands-on experience programming 3D graphics applications under Linux. It is one thing to understand the theoretical mechanics of an algorithm; it is another to actually implement, debug, and optimize that same algorithm using a particular set of programming tools. Small standalone programs are scattered throughout this text to demonstrate key 3D graphics concepts. It is often easy to lose sight of the forest for the trees, particularly in the complicated world of 3D graphics. Standalone sample programs address this problem by concisely illustrating how all the necessary components of a 3D program "fit together." They reduce the intimidation that often accompanies the study of large, complicated programs, and give you confidence in developing and modifying complete 3D programs under Linux.

A third goal of this text is to help you develop and understand the techniques for developing a reusable 3D application framework or library. In addition to the standalone programs mentioned above, the book also develops a series of generally reusable C++ library classes for 3D graphics, called the l3d library. This library was introduced in the introductory companion book *Linux 3D Graphics Programming* and is developed further in this book. This C++ library code follows an object-oriented approach, relying heavily on virtual functions, (multiple) inheritance, and design patterns. In this manner, the developed library classes are usable as is but still open for extension through subclassing. Each chapter builds upon the library classes developed in previous chapters, either adding new classes or combining existing classes in new ways. Through subclassing, the library classes can be adapted to work with virtually any hardware or software platform or API; currently, the code runs under Linux and Microsoft Windows, with or without hardware acceleration. The techniques used to develop the 3D library classes illustrate both valuable 3D abstractions and generally applicable object-oriented techniques.

A fourth goal of this text is to demonstrate the excellence of the Linux platform as a graphics programming environment. For a programmer, Linux is a dream come true. All of the source code is available, all of the operating system features are enabled, a large number of excellent first-rate software development tools exist, and it is all freely available, being constantly tested and improved by thousands of programmers around the world. Linux empowers the programmer with open source, open information, and open standards. Given this outstanding basis for development, it is no wonder that programmers in every conceivable application area (including 3D graphics) have flocked to Linux. This has created a wealth of 3D libraries, tools, and applications for Linux. Linux is, therefore, an outstanding software development platform with powerful 3D tools and software—an ideal environment for learning and practicing 3D graphics programming.

A final, personal goal of this text, and the main reason I am writing this book, is to impart to you a sense of the excitement that 3D graphics programming offers. You, the 3D programmer, have the power to model reality. You control every single *z*-buffered, Gourad-shaded, texture-mapped, perspective-correct, dynamically morphed, 24-bit, real-time pixel on the flat 2D screen, and amazingly, your painstakingly coded bits and bytes merge to form a believable 3D

world. By working under Linux, you are no longer held back by a lack of tools or software. It's all out there—free for download and top quality. Linux software gives you the tools you need to realize your 3D ideas.

Organization of the Book and the Code

This text follows a bottom-up organization for the presentation order of both concepts and program code. This bottom-up organization serves two purposes: pedagogical and practical.

Seen pedagogically, a bottom-up approach means first covering fundamental concepts before proceeding to more complex subjects. This is a fully natural progression which deals with computer graphics at ever-increasing levels of abstraction. Seen practically, a bottom-up approach means that simple C++ classes are developed first, with later, more complicated examples literally "building upon" the foundation developed earlier through the object-oriented mechanism of inheritance. This ensures compilable, executable code at each level of abstraction which is incrementally understandable and extensible. Every chapter has complete, executable sample programs illustrating the concepts presented.

The bottom-up organization has a rather far-reaching impact on the structure of the code in general. The principal goal I had in mind when structuring the code for the book was that all parts of a class presented in a chapter should be explained within that same chapter. I tried very diligently to achieve a code structure which allows me to avoid statements like "ignore this part of the code for now; it will be explained in the next chapter." When a class is presented, you should be able to understand it fully within the context of the current chapter. The second most important goal for the code was to reuse as much code as possible from previous chapters, typically through subclassing, thus truly illustrating how more complex 3D concepts literally, at the code level, build upon simpler concepts. To achieve these goals, the overall design of the code relies heavily on indirection through virtual functions, even in fairly time-critical low-level routines such as accessing elements of a list. The presence of so many virtual functions allows for a rather clean, step-by-step, bottom-up, incrementally understandable presentation of the code. The design is also very flexible; new concepts can be implemented through new subclasses, and behavior can be swapped out at run time by plugging in new concrete classes. But as is always the case in computer science, there is a tradeoff between flexibility and performance. The code design chosen for the book is not as fast as it could be if all the virtual function calls were eliminated; of course, eliminating virtual function calls leads to reduced flexibility and increased difficulty extending the code later. Still, the code performs well: it achieves over 30 frames per second with software rendering on a Pentium II 366 in a 320 240 window with 24-bit color, and over 30 frames per second in 1024 768 with Voodoo3 hardware acceleration. In spite of its definite educational slant, it is fast enough for real use. Again, this is one of the great things about 3D programming in the 21st century: a straightforward, educationally biased code structure can still be executed fast enough by consumer hardware for real-time, interactive 3D environments. Real-time 3D no longer forces you to wrestle with assembly or to have access to expensive dedicated graphics workstations. If you know how to program in C++ and you understand the geometrical concepts behind 3D graphics, you can program real-time 3D graphics applications using free tools under Linux.

Let's now look at the organization of the text itself.

Chapter 1 reviews the essentials of Linux 3D graphics, as covered in the introductory companion volume *Linux 3D Graphics Programming*. We cover the fundamentals of 2D graphics, 3D coordinate systems, perspective projection, vectors, matrices, and the C++ library classes—the l3d library—used to implement these basic ideas. We also review the most important commands for the 3D modeling package Blender. The information in this chapter is a prerequisite for understanding the rest of the book.

Chapter 2 explores some important techniques which greatly increase the visual realism of polygonal models: texture mapping, lighting, light mapping, and morphing. All of these techniques are implemented in C++ classes. We also take a tour of the symbolic algebra package Calc, available as an extension to the Emacs editor. Calc helps us solve the tedious sets of equations which arise when performing texture mapping.

Chapter 3 is the first of two chapters dealing with Blender, a free and powerful 3D modeling and animation package for Linux (included on the CD-ROM). In two step-by-step tutorials, we walk through the creation of a set of textured and compatible morph targets suitable for 3D morphing, and a human-like figure animated with inverse kinematics. These 3D models are then imported and displayed in a 3D program.

Chapter 4 deals with the correct and efficient drawing of visible surfaces, which becomes especially important when polygon counts increase. Surfaces which are obscured by other surfaces or which are completely outside of the field of vision should be discarded from unnecessary processing as early and as cheaply as possible. We discuss a number of generally applicable techniques, each illustrated with a sample program: back-face culling, the painter's algorithm, view volume culling, and *z*-buffering.

Chapter 5 discusses special visible-surface algorithms based on space-partitioning techniques. The techniques discussed include BSP trees, octrees, regular spatial partitioning, and portals. We discuss the use of portals for special techniques such as mirrors, refraction, transparency, and volumetric shadows. A portal engine is implemented as a natural extension of the existing polygon and object classes.

Chapter 6 continues the discussion of portals from a practical point of view. We explore how we can use Blender's powerful modeling features to create portal-based worlds, using a combination of an edge-coding technique, to encode arbitrary data within a 3D mesh, and postprocessing scripts written in the Perl language. This system, using both Blender and custom-written tools, allows us to create 3D worlds which may be used by the portal engine developed in the previous chapter. A complete example world is constructed step by step, with practical tips on efficiently working in Blender: using layers, hiding geometry, aligning objects and portals, and executing interactive fly-throughs.

Chapter 7 covers some special 3D graphics techniques or "tricks": billboards, lens flare, particle systems, fractal landscapes, dynamic level-of-detail, environment mapping, atmospheric effects, curved surfaces, multi-pass techniques, and camera tracking in 3D.

Chapter 8 discusses non-graphical techniques that can greatly enhance the reality of 3D graphics programs, such as games. Techniques discussed and implemented include: collision detection, digital sound and music with the RPlay sound server, TCP/IP network communications, physics, and artificial intelligence.

Chapter 9 takes a look at the possible future direction of Linux and 3D graphics. We begin with a look at two existing and exciting 3D content development systems under Linux: Game Blender and World Foundry. We go through a brief tutorial of 3D game creation with each of these systems. Some speculation about the future of Linux 3D graphics follows. We close by relating the contents of the book to the field of 3D graphics as a whole.

The Appendix provides installation instructions for the CD-ROM, information on porting the graphics code to Windows, and a list of useful references, both in electronic (WWW) and in print form. Notations in brackets, such as [MEYE97], are detailed in the "References" section of the Appendix.

The CD-ROM contains all sample code from the book, the Blender 3D modeling and animation suite, the World Foundry game development kit, freely available Linux 3D libraries and applications, and a series of animated videos illustrating some of the more difficult-to-visualize 3D concepts discussed in the text.

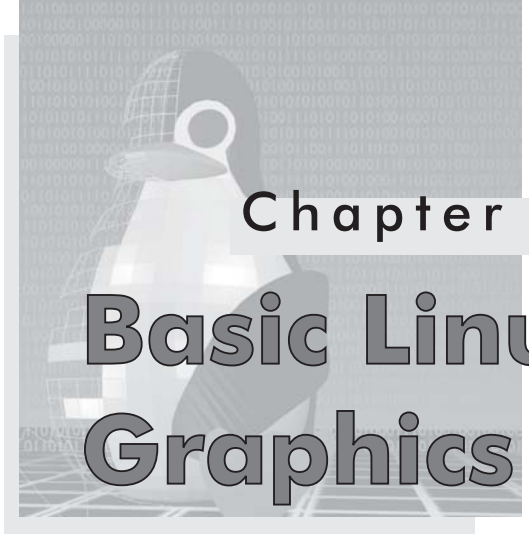
Reader and System Requirements

This book requires you to have a working Linux installation up and running with the XFree86 server for the X Window System on an IBM PC or compatible system with a Pentium or better processor. If you don't yet have Linux installed, you can download Linux for free from the Internet, or obtain a CD-ROM containing a ready-to-install Linux distribution. Installing Linux is no more difficult than installing other common PC operating systems, such as Microsoft Windows. A 3D graphics card with Mesa drivers is recommended for optimum performance, but the code will run acceptably fast without hardware acceleration through a custom software renderer. If your graphics card is supported by the new XFree86 4.0 Direct Rendering Infrastructure or by the Utah GLX project, you can also link the code with the appropriate OpenGL library (from the DRI or from Utah GLX) to achieve hardware-accelerated rendering in a window.

Typographical Conventions Used in This Book

The following typographical conventions are used in this book.

- Program code, class names, variable names, function names, filenames, and any other text identifiers referenced by program code or the operating system are printed in a `fixed-width font`.
- Commands or text to be typed in exactly as shown are printed in **boldface**.
- Key sequences connected by a plus (+) sign (such as Ctrl+C) mean to hold the first key while typing the second key.



Chapter 1
Basic Linux 3D Graphics Concepts

Linux 3D graphics is a young and exciting field. The purpose of this chapter is to review the basic concepts of Linux 3D graphics programming in order to lay a groundwork for the more involved material in the following chapters. This book was written based on the assumption that you already know everything in this chapter; therefore, this chapter is intentionally terse. This chapter is meant to serve as a review, not as an introduction.

If you find some of these topics unfamiliar, I suggest that you take the time to read the companion volume to this book, *Linux 3D Graphics Programming*. The companion book is aimed at the beginning 3D graphics programmer with little or no 3D experience. Essentially, this chapter is a brief review of the most important concepts in the introductory companion volume.

2D Graphics Fundamentals

2D raster graphics consist of plotted pixels on a display. The pixels are arranged in a rectangular grid, typically accessible in memory as a linear sequence of bytes. Though we specify pixels by their addresses in memory at the lowest level, it is better to specify pixels in terms of a 2D coordinate system, with horizontal x and vertical y axes. In this book, we define the origin of the 2D pixel coordinate system to be the upper-left corner of the screen, with the x axis increasing to the right and the y axis increasing downward.

Under Linux, we display our graphics under the X Window System. Specifically, the approach chosen for this book is to use XImages in ZPixmap format to display 2D graphics. This allows us direct access to the bytes (and thus the pixels) forming the image. Each pixel can have a particular color. Exactly how this color is specified depends on the bit depth and color model of the X server. The bit depth determines the total number of available colors and is usually 8, 15, 16, 24, or 32 bits. The color model is typically either indexed color (meaning that colors are specified as indices into a fixed-size palette of colors) or true color (meaning that colors are specified directly as a combination of red, green, blue, and possibly alpha intensities). For maximum flexibility, we must query at run time the bit depth and color model, and dynamically determine the exact bit format required to specify pixel colors.

Drawing lines can be done by an incremental algorithm, stepping along one axis by whole pixels and using the line's slope to determine how many pixels to step in the other axis. Drawing polygons can be done by rasterizing the lines belonging to the left and right edges of the polygon, and drawing horizontal lines, or *spans*, between the left and right edges.

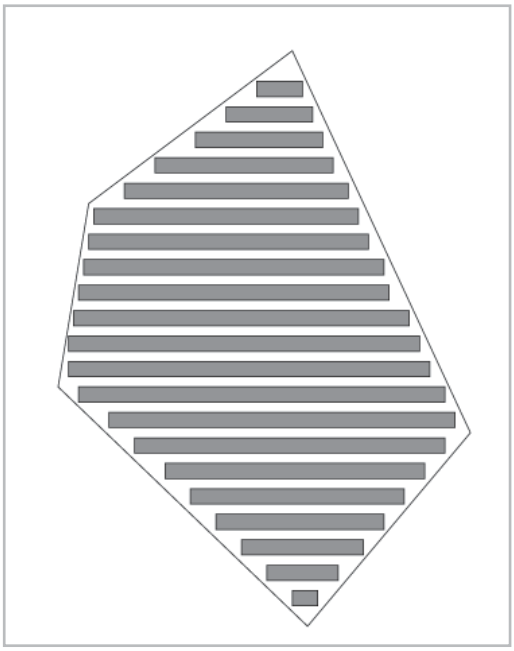


Figure 1-1: Drawing a 2D polygon.

Animation can be achieved through double buffering. With this technique, we have one off-screen buffer and one on-screen buffer. We draw graphics in the off-screen buffer. When we are finished, we copy the off-screen buffer into the on-screen buffer, at which point the graphics become visible. Then, we draw the next frame of animation in the off-screen buffer, and repeat the process. In this way, the on-screen buffer is continually updated with new and completed images from the off-screen buffer, thus creating the illusion of animation.

Hardware acceleration allows us to send compact instructions to dedicated hardware, with higher-level commands such as "draw a line" or "draw a polygon." By sending such higher-level commands to the hardware, and by letting the dedicated hardware then do the actual lower-level pixel operations, a great speed-up can be achieved. Under Linux, we use the 3D library Mesa to achieve hardware acceleration. Mesa has a syntax essentially identical to OpenGL and supports hardware acceleration. The XFree86 4.0 project uses Mesa as part of its Direct Rendering Infrastructure (DRI), providing for hardware-accelerated 3D graphics within a window under the X Window System. We use the terms Mesa and OpenGL essentially interchangeably in this book.

3D Graphics Fundamentals

3D graphics is the creation of a two-dimensional image or series of images on a flat computer screen such that the visual interpretation of the image or series of images is that of a three-dimensional image or series of images.

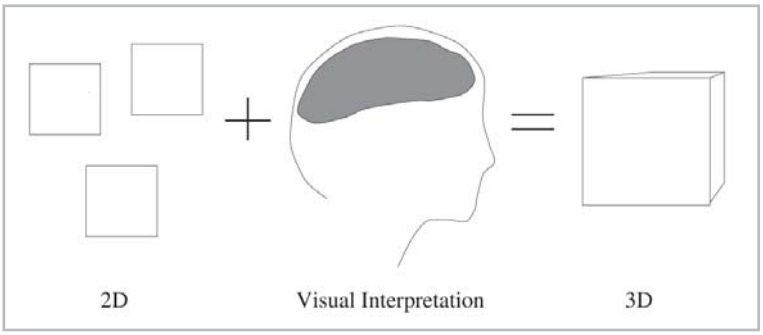


Figure 1-2: Definition of 3D graphics.

The visual interpretation of an image depends on the optics of light rays striking the retina. Points emit light radially in all directions along straight lines, called *light rays*. Some subset of all light rays enters the eye; we call this subset *seen light rays*. Seen light rays are refracted by the eye's lens, and focus onto the retina. The biochemical reactions within the retina produce the sensation of vision.

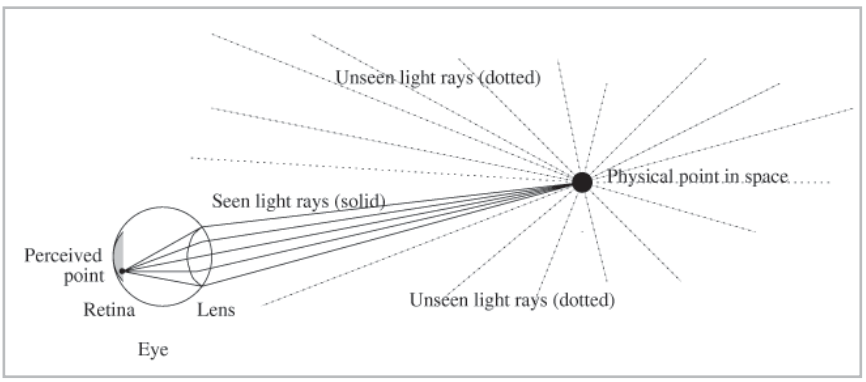


Figure 1-3: Light rays.

If a 2D image causes the same light rays to strike the retina as those coming from a 3D object, then the 2D image can be interpreted as a 3D object.

In 3D graphics, we want the seen light rays coming from the flat computer screen to correspond to the seen light rays which would be seen if we were looking at a real 3D object located behind the computer screen. To accomplish this, we compute intersections between the seen light rays and the flat plane of the computer screen. These intersection points, since they lie along the straight light rays going from the original 3D points to the eye, emit the same seen light rays as the

original points. Therefore, the projected points can be visually interpreted to be the original 3D object.

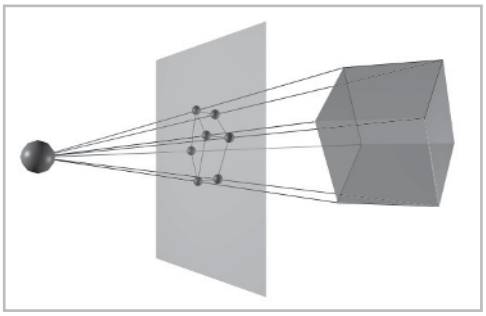


Figure 1-4: A 2D image can be interpreted as a 3D object.

TEAMFLY

Computing an intersection between a seen light ray (coming from a point) and a flat plane is called *projecting* the point onto the plane. In particular, this is a *planar geometric perspective projection*. The important term is "perspective." The fact that the projection is perspective implies that the resulting images appear realistically foreshortened, just as would be perceived by our own eyes or by a physical camera taking a 2D snapshot of a 3D scene.

3D Coordinate Systems and Vectors

Before we can perform a perspective projection on points, we need a *coordinate system* to specify the points in 3D. We can use a left-handed or a right-handed coordinate system. Define a coordinate system such that $x \times y = z$, where the symbol $\times$ represents the vector cross product. In a right-handed system, the vector cross product is computed using the right-handed convention, implying that in Figure 1-5, the z axis points out of the page. In a left-handed system, the vector cross product is computed using the left-handed convention, implying that in Figure 1-6, the z axis points into the page.

In this book, we use the left-handed coordinate system, and the left-handed convention for computing our vector cross products. By using both a left-handed coordinate system and a left-handed rule for computing cross products, the results we obtain and the equations we use are identical to those which would be used with a right-handed coordinate system and a right-handed rule for computing cross products.

Within a coordinate system, we can specify points and vectors. *Points* are locations in the coordinate space; *vectors* are directed displacements between points in the coordinate space. Be careful not to confuse points and vectors. One way to specify points and vectors is to use the notation (x,y,z). (Another way is homogeneous notation; see the next section.) In this notation, the point (x,y,z) is the point located at a distance of x units from the origin along the x axis, y units from the origin along the y axis, and z units from the origin along the z axis. On the other hand, the vector (x,y,z) specifies the displacement of x units along the x axis, y units along the y axis, and z units along the z axis. Since it is a directed displacement, we can add the vector (x,y,z) to any point to arrive at a new, displaced point.

Team-Fly®

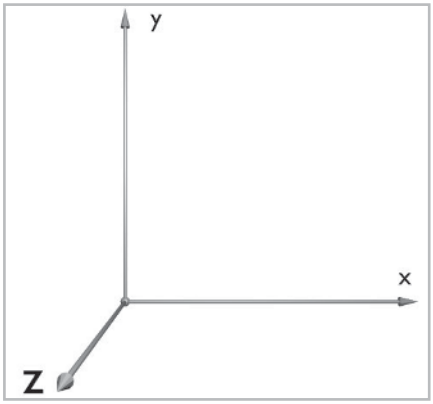


Figure 1-5: Right-handed 3D coordinate system.

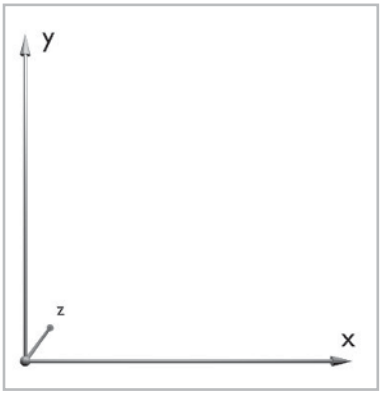


Figure 1-6: Left-handed 3D coordinate system.

A number of standard operations are defined on vectors. Important for this book are component-wise vector addition, scalar-vector multiplication, the vector dot product, and the vector cross product.

Equation 1-1 $$vector\ v + vector\ w = (v_1, v_2, v_3) + (w_1, w_2, w_3) = (v_1 + w_1, v_2 + w_2, v_3 + w_3)$$

Equation 1-2 $$vector\ v \cdot scalar\ s = (v_1, v_2, v_3) \cdot s = (s \cdot v_1, s \cdot v_2, s \cdot v_3)$$

Equation 1-3 $$vector\ v \cdot vector\ w = (v_1, v_2, v_3) \cdot (w_1, w_2, w_3) = (v_1 w_1 + v_2 w_2 + v_3 w_3)$$

Equation 1-4 $$v \times w = (v_2 w_3 - v_3 w_2, v_3 w_1 - v_1 w_3, v_1 w_2 - v_2 w_1)$$

Perspective Projection

Given a coordinate system in which to specify (*x*,*y*,*z*) points, we can then apply a perspective projection to these points to obtain the projected points, for which the seen light rays are identical to those of the original non-projected points. In its simplest form, the perspective projection of a point (*x*,*y*,*z*) is:

Equation 1-5 $$\begin{aligned} x_p &= d\frac{x}{z} \\ y_p &= d\frac{y}{z} \end{aligned}$$

This formula is derived by computing the intersection between the seen light ray, coming from the point, and a flat 2D projection plane. The *d* term is essentially a scaling factor. A more complete formulation of the perspective projection is:

Equation 1-6 $$\begin{aligned} x_s &= (\tfrac{x}{z}(0.5\cot(0.5\theta)) + 0.5)w_s \\ &= \tfrac{x}{z}(0.5\cot(0.5\theta)w_s) + 0.5w_s \\ y_s &= (-\tfrac{y}{z}(0.5\cot(0.5\theta)) \cdot \tfrac{w_{Phys}}{h_{Phys}} + 0.5scale_y)h_s \\ &= -\tfrac{y}{z}(0.5\cot(0.5\theta) \cdot \tfrac{w_{Phys}}{h_{Phys}} \cdot h_s) + 0.5scale_y h_s \end{aligned}$$

This form of the equation explicitly specifies the use of the *d* term as a field of view angle theta. Also, it reverses the *y* axis orientation because it maps the projected points to the 2D pixel

coordinate system. As we have seen, the 2D pixel coordinate system has *y* increasing downwards, while the 3D coordinate system has *y* increasing upwards.

Matrices

In this book, we use 4 4 matrices to effect transformations on points. We can also then use 4 1 column vectors to represent 3D points and 3D vectors. Do not confuse the terms "column vector" and "3D vector." The former refers to a notational convention; the latter, to a directed displacement in 3D space. The latter can be expressed by using the notation provided by the former. A 3D point expressed in column vector notation is $[x,y,z,1]^T$. A 3D vector expressed in column vector notation is $[x,y,z,0]^T$. The superscripted "T" indicates that the vectors should actually be written transposed, in a vertical format. The fourth coordinate is *w*, the homogeneous coordinate; points and vectors expressed in this notation are said to be in *homogeneous coordinates*. The homogeneous *w* coordinate typically has a value of 1 for points and 0 for vectors. In general, for any arbitrary non-zero value of *w*, the homogeneous point $[x,y,z,w]^T$, corresponds to the location in 3D space given by $[x/w,y/w,z/w,1]^T$. In other words, we divide by *w*.

We multiply two matrices *A* and *B*, with a result called *C*, as follows. Treat each column of *B* as a four-element vector. Treat each row of *A* as a four-element vector. Then, compute the value of each element in resultant matrix *C* located at row *i* and column *j* as the dot product of row *i* in *A* and column *j* in *B*. Not all matrices may be multiplied with one another; the definition of matrix multiplication implies that the matrices to be multiplied must be size-compatible. Also, in general, matrix multiplication is not commutative; *AB* is generally not the same as *BA*—the multiplication *BA* might not even be possible.

Multiplying a 4 4 matrix by a second 4 4 matrix yields a resulting 4 4 matrix whose transformation is the concatenation of the transformations represented by the first two matrices. The resulting composite transformation applies the transformation of the original right-hand matrix first, followed by the transformation of the original left-hand matrix. (An alternative interpretation, using a changing-coordinate system view rather than a changing-point view, allows for a left-to-right interpretation of the transformation order.) Multiplying a 4 4 matrix by a 4 1 matrix (in other words, by a column vector representing a 3D point or a 3D vector) yields another 4 1 matrix which represents the 3D point or 3D vector transformed by the 4 4 matrix.

Specific Matrix Transformations

The matrix forms of several important 3D transformations follow.

Equation 1-7
Rotation around the x axis by degrees

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 1-8
Rotation around the y axis by degrees

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 1-9
Rotation around the z axis by degrees

$$R_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 1-10
Scaling by factors sx, sy, and sz in the x, y, and z axes

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 1-11
Translation by an offset of (tx,ty,tz)

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 1-12
Transformation from camera space to world

$$M_{world \leftarrow camera} = \begin{bmatrix} VRI_x & VRI_y & VRI_z & 0 \\ VUP_x & VUP_y & VUP_z & 0 \\ VFW_x & VFW_y & VFW_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -VRP_x \\ 0 & 1 & 0 & -VRP_y \\ 0 & 0 & 1 & -VRP_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In the equation above, the camera is located at (VRPx,VRPy,VRPz) and is oriented with its right-vector along (VRIx,VRIy,VRIz), its up-vector along (VUPx,VUPy,VUPz), and its forward-vector along (VFWx,VFWy,VFWz).

Equation 1-13
Rotation by degrees about an arbitrary vector (u1,u2,u3)

$$\begin{bmatrix} u_1^2 + cos\theta(1 - u_1^2) & u_1u_2(1 - cos\theta) - u_3sin\theta & u_3u_1(1 - cos\theta) + u_2sin\theta & 0 \\ u_1u_2(1 - cos\theta) + u_3sin\theta & u_2^2 + cos\theta(1 - u_2^2) & u_2u_3(1 - cos\theta) - u_1sin\theta & 0 \\ u_3u_1(1 - cos\theta) - u_2sin\theta & u_2u_3(1 - cos\theta) + u_1sin\theta & u_3^2 + cos\theta(1 - u_3^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Other Matrix Properties

The inverse of a matrix is the matrix which, when multiplied with the original matrix, yields the identity matrix *I*. The identity matrix is a square matrix with all zero entries except for a series of entries with value 1 located along the main diagonal of the matrix, from the upper-left to the lower-right corner. When viewing matrices as transformations, the inverse of a matrix then represents the opposite of the transformation represented by the original matrix. We denote the inverse of a matrix *M* as M^{-1}.

A 4 4 matrix can be viewed as a specification of a coordinate system. The first three columns of the matrix represent the *x*, *y*, and *z* axes of the coordinate system. The last column of the matrix represents the origin point of the coordinate system. By multiplying a point with this matrix, we obtain the world coordinates of the point as seen relative to the coordinate system of the matrix. In other words, if we have a matrix *M* and a point *P*, then in the matrix product *MP*, the matrix *M* represents the coordinate system in which P is specified. The product *MP* yields the location of the *P*

in the world coordinate system. By inverting a matrix representing a coordinate system, we obtain the reverse transformation. Therefore, the matrix product $M^{-1}P$ yields the coordinates relative to M of the point as specified in the world coordinate system.

When you see the matrix product MP, think of this as answering the question "P, which has been specified relative to M, is at what location in world coordinates?" When you see $M^{-1}P$, think of this as answering the question "P, which has been specified in world coordinates, is at what location relative to M?"

The l3d Library Classes

This book relies on the use of a series of C++ library classes implementing all of the 2D and 3D graphics concepts described in the previous sections. This library is called the l3d library. It is developed incrementally in the introductory companion book, *Linux 3D Graphics Programming*. In this book, we use the classes presented in the first book, and continue to build on these classes to illustrate newer and more advanced concepts. The l3d classes are on the CD-ROM and are also available for download from the Internet at `http://www.linux3dgraphics-programming.org`.

Sample l3d Program

Before looking at the l3d classes themselves, let's first look at a sample program which uses l3d. This will give you a practical perspective on l3d before looking at the following sections, which go into more detail on the specific l3d classes.

The following sample program is called `drawdot` and illustrates usage of the l3d library classes in order to move a green dot around the screen, thereby forming a simple drawing program. This program works with visuals of any color depth and in both TrueColor or indexed color modes. Notice that this program is rather short and declares only one class. This is because the l3d library has already declared several useful classes to simplify application programs.

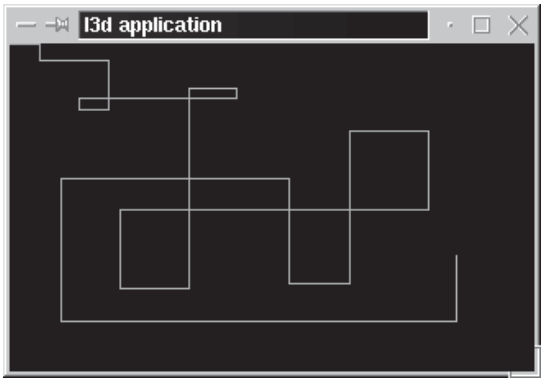


Figure 1-7: Output from sample program `drawdot`.

Listing 1-1: drawdot.cc

```
#include <stdlib.h>
#include <stdio.h>

#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rasteriz.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/system/factorys.h"

//-----------------------------------------------------------------------
//-
//- STEP 1: CHOOSE THE FACTORIES
//-
//-----------------------------------------------------------------------

void choose_factories(void) {
  factory_manager_v_0_1.choose_factories();
}

//-----------------------------------------------------------------------
//-
//- STEP 2: DECLARE A PIPELINE SUBCLASS
//-
//-----------------------------------------------------------------------

class my_pipeline : public l3d_pipeline {
  protected:
    l3d_rasterizer_2d_imp *ri;
    l3d_rasterizer_2d *r;

    int x, y, dx, dy;
    unsigned long color;

  public:
    l3d_screen *s;
    my_pipeline(void);
    virtual ~my_pipeline(void);

    void key_event(int ch);  //- from dispatcher
    void update_event(void); //- from dispatcher
    void draw_event(void);   //- from dispatcher
};

my_pipeline::my_pipeline(void) {
  s = factory_manager_v_0_1.screen_factory->create(320,200);
  ri = factory_manager_v_0_1.ras_2d_imp_factory->create(320,200,s->sinfo);
  r = new l3d_rasterizer_2d(ri);

  s->sinfo->ext_max_red =
    s->sinfo->ext_max_green =
      s->sinfo->ext_max_blue =  255;

  s->sinfo->ext_to_native(0, 0, 0); //- allocate background color
  color = s->sinfo->ext_to_native(0, 255, 128);
  s->refresh_palette();

  x = y = dx = dy = 0;
}
```

```
my_pipeline::~my_pipeline(void) {
  delete s;
  delete ri;
  delete r;
}

void my_pipeline::key_event(int ch) {
  switch(ch) {
    case 'h': dx=-1; dy=0; break;
    case 'l': dx=1; dy=0; break;
    case 'j': dx=0; dy=1; break;
    case 'k': dx=0; dy=-1; break;
    case ' ': dx=0;dy=0; break;
    case 'q': {
        exit(0);
      }
  }
}

void my_pipeline::update_event() {
  x += dx;
  y += dy;

  if(x < 0) x = 0;
  if(x > s->xsize-1) x = s->xsize-1;
  if(y < 0) y = 0;
  if(y > s->ysize-1) y = s->size-1;
}

void my_pipeline::draw_event(void) {
  r->draw_point(x,y, color);
  s->blit_screen();
}

main() {
  choose_factories();

  l3d_dispatcher *d;
  my_pipeline *p;

  //-----------------------------------------------------------------------
  //-
  //- STEP 3: CREATE A DISPATCHER
  //-
  //-----------------------------------------------------------------------

  d = factory_manager_v_0_1.dispatcher_factory->create();

  //-----------------------------------------------------------------------
  //-
  //- STEP 4: CREATE A PIPELINE
  //-
  //-----------------------------------------------------------------------

  //- plug our custom behavior pipeline into the dispatcher
  p  = new my_pipeline();

  //-----------------------------------------------------------------------
  //-
  //- STEP 5: START DISPATCHER
```

```
  //-
  //------------------------------------------------------------------------

  d->pipeline = p; //- polymorphic assignment
  d->event_source = p->s;
  d->start();

  delete d;
  delete p;
}
```



NOTE The following instructions assume you have already installed the source code as described in the Appendix. In particular, the installation instructions require you to set the `$L3D` environment variable to point to the installation directory. Also, you should execute these commands in a shell window under the X Window System. See the Appendix for details on initial installation and configuration of the source code.

First, let's look at compiling the program. Then, we look at the structure of the program itself. Finally, we discuss the l3d classes.

The source code for the sample program is located in directory `$L3D/source/app/drawdot`. The source and binary files are located in different directory trees; the next section discusses this in detail. For now, compile and run the program as follows:

1. To compile the l3d library, type **cd $L3D/source/app/lib**, press **Enter**, type **make -f makeall.lnx**, and press **Enter**. Notice that this makefile has a different filename than the standard name of `Makefile`; therefore, we specify the `-f` flag to tell the `make` command which file is the makefile.
2. Change to the source directory for `drawdot` by typing **cd $L3D/source/app/drawdot** and press **Enter**. Compile `drawdot`: type **make -f makeall.lnx** and press **Enter**.
3. Type **cd $L3D/binaries/linux_x/float/app/drawdot** to change to the binaries directory and press **Enter**.
4. Notice the object and executable files from the compilation process are placed in the corresponding binary directory. Type **drawdot** and press **Enter** to run the program.
5. Notice the question "which configuration?" Type **1** for now to select a normal X11 window, and press **Enter**.
6. Notice the empty black window which appears. Type **l**. Notice the green line which moves from left to right across the very top of the display, and that the line continues moving after you release the key.
7. Type **j**. Notice that the green line moves downward.
8. Control the movement of the line with the following keys: **h** to move left, **l** to move right, **j** to move down, **k** to move up, and **Space** to stop movement.
9. Type **q** to end the program.

Having now successfully executed the program, let's now take a look at the organization of the l3d directory structure, then examine the `drawdot` program itself.

l3d Directory Structure

The library classes and the applications using the library classes are split into separate source and binary directory trees. The reason we split the source files and binary files into two separate trees is that the same source code can be compiled on a number of different platforms. Keeping the binary files, which vary from platform to platform, in the same directory as the source files, which remain the same, leads to a rather chaotic organization. While chaos theory is a rich and fascinating field of scientific study, we don't necessarily want to organize our directory structure on this basis. Splitting source and binary directories allows for a neater multi-platform directory structure.

Specifically, the following directory structure is used:

- `$L3D/source`: All source files.
- `$L3D/source/util`: Non-C++ source files (preprocessing scripts, etc.)
- `$L3D/source/app`: C++ source files related directly to 3D applications.
- `$L3D/source/app/lib`: C++ source for the l3d library classes.
- `$L3D/source/app/[program_name]`: C++ source for example programs. A few simple programs place the binary files in source directory, but most programs place them in the binary directory.
- `$L3D/binaries/linux_x`: Linux binary files compiled for the X Window System.
- `$L3D/binaries/linux_x/fixed`: Linux binary files compiled with fixed-point math. Fixed-point executable are currently not very well supported and do not always function correctly. Subdirectory structure is the same as that under the float subdirectory.
- `$L3D/binaries/linux_x/float`: Linux binary files compiled with floating-point math. This is the primary output directory for binary files.
- `$L3D/binaries/linux_x/float/app/lib`: Linux floating-point binary files for the l3d library.
- `$L3D/binaries/linux_x/float/app/[program name]`: Linux floating-point binary files for example programs.

The makefiles automatically place the binary files in the corresponding binary directory. You typically invoke **make -f makeall.lnx** in the source directory for an application program, which then compiles all of the Linux binaries and places them in the appropriate binaries directories.



NOTE Remember, the Appendix provides instructions on how to compile all of the sample programs at once. The preceding discussion is primarily to give you an idea of the directory structure and the reasoning behind it.

To summarize, then, the source files for the l3d library are in `$L3D/source/app/lib`, and the source files for the sample programs are all in `$L3D/source/app`. The primary binaries are in `$L3D/binaries/linux_x/float/app`.

The Five-Step Process of l3d Programs

The `drawdot` program can be broken up into five steps, which are representative of programming with l3d. In fact, these steps are representative of event-based programming in general, on a variety of platforms, as evidenced by the fact that the following scheme also can be applied to event-driven programs under various operating systems.

The five steps are as follows.

1. *Choose the proper factories* for three classes: the screen, the rasterizer implementation, and the event dispatcher.
2. *Declare a pipeline subclass*. The pipeline must directly or indirectly ask the factory to create a screen and a rasterizer implementation (typically in the pipeline constructor). Override the abstract pipeline methods to allow your program to respond to events, to update itself, and to draw to the screen using the rasterizer implementation.
3. *Create a dispatcher* by using the factory.
4. *Create a pipeline*. Your pipeline should in its constructor ask the factory to create a screen and a rasterizer implementation, and store these objects locally. Connect the pipeline and the screen to the dispatcher.
5. *Start the dispatcher.* The dispatcher enters an event loop, extracts events from the screen, and calls your pipeline periodically to allow your pipeline to do its work, respond to input, and draw to the screen.

Let's examine each step in detail to understand the general l3d structure within the context of the sample `drawdot` program. This serves two goals: first, to understand the general l3d structure, and second, to understand the specific functions called by `drawdot` in order to draw to the screen. Then, we will take a look at the l3d classes themselves, which we build upon throughout the book to incorporate increasingly advanced and reusable 3D graphics concepts.

Step 1. Choose the Proper Factories

The first step in writing an l3d application is to choose the proper factories for the program. This is done by calling the `choose_factories` function defined in the so-called *factory manager*. To localize object creation and free applications from needing to know specific details of concrete classes, we use the *factory design pattern*. The factory manager is the central location where all concrete factories, which are globally visible to the entire program, are accessible. The following line chooses the factories within the factory manager:

```
factory_manager_v_0_1.choose_factories();
```



NOTE The class name has the suffix `v_0_1` to represent the fact that this is the first version of the factory manager. Later versions of the factory manager class manage more factories. This is an example of how subclassing can provide a historical record of program development, within the source code itself.

Choosing the factories essentially means customizing, at run time, all customizable behavior, which then takes effect for the duration of the program. In particular, the l3d factory manager manages three factories:

1. A screen factory, producing objects corresponding to the abstract interface `l3d_screen`. Class `l3d_screen` represents the physical output device—a window under X11, a full-screen hardware-accelerated window using Mesa, or even a DIBSection under Microsoft Windows.
2. A rasterizer implementation factory, producing objects corresponding to the abstract interface `l3d_rasterizer_2d_imp`. Class `l3d_rasterizer_2d_imp` represents a particular implementation of 2D rasterization concepts. We use the term "rasterizer" to denote a software interface to a rasterizer implementation. A rasterizer implementation is a particular hardware or software component that draws 2D graphics primitives (triangles, lines, dots) into a frame buffer. Two important types of rasterizer implementations are software rasterizer implementations, which write directly into an off-screen buffer, and hardware rasterizer implementations, which use specialized, faster functions for hardware-accelerated pixel operations.
3. A dispatcher factory, producing objects corresponding to the abstract interface `l3d_dispatcher`. Class `l3d_dispatcher` represents a generalized event dispatcher in a particular operating system environment. Under X, the dispatcher intercepts X events within a window's event loop and passes them on transparently to our application. Using hardware acceleration with Mesa, the dispatcher works within the event framework provided by Mesa and GLUT, again forwarding events in a transparent way to our application. Under another operating system, the dispatcher would need to call any OS-specific routines necessary to capture and forward events.

All of these factories represent system-specific information: the output device, the rasterizer implementation, and the event dispatcher. Therefore, by choosing the factories, we are essentially dynamically configuring the program to use the desired run-time environment. In our case, the factory manager simply asks the user which factories should be used, but more sophisticated solutions are also possible. We could, for instance, have an auto-detect routine that searches for the existence of particular hardware, and that, depending on whether or not it finds it, configures the factory to create the appropriate software component accordingly.

Step 2. Declare a Pipeline Subclass

The second step in writing an l3d application is to declare a *pipeline* subclass. A pipeline is simply a sequence of operations on data. The main loop in a game or graphics program is typically called the pipeline. Therefore, the pipeline contains, directly or indirectly, your application's main data and functionality.

We say "directly or indirectly" because the pipeline might do nothing other than create another object, to which it then delegates the main program's responsibility. In such a case, the pipeline is not directly responsible for the application's data and functionality, but instead merely serves as an interface between the dispatcher and the object actually doing the real work.

Team-Fly®

The pipeline does not control execution of the program. Instead, it responds to events. The abstract `l3d_pipeline` class provides a set of virtual event functions, which are automatically called by an event dispatcher (class `l3d_dispatcher`, covered in the next section). By declaring a subclass of `l3d_pipeline`, you can override the virtual event functions to provide specific responses to specific events, without needing to know how or when these functions are invoked.

In particular, an `l3d_pipeline` subclass should do three things:

1. Directly or indirectly create and store a screen object, a rasterizer implementation object, and a rasterizer object. This is typically done in the constructor. The first two objects, the screen and rasterizer implementation, must be created by using the already chosen factories (section "Step 1: Choose the Proper Factories"). The third object, the rasterizer itself, is directly created via the C++ operator `new`, since the rasterizer itself contains no platform-specific dependencies. (Such dependencies are all in the rasterizer implementation, not the rasterizer.)
2. Declare internal variables, functions, and objects to store the current state of the virtual world.
3. Override the `l3d_pipeline` virtual event functions to handle input, update internal objects, and draw output to the screen. Handling input and updating internal objects are both done by using data structures specific to the application program. Drawing output to the screen is done by using the screen, rasterizer, and rasterizer implementation objects created in the constructor.

The first responsibility of an `l3d_pipeline` subclass is easy to understand. The pipeline represents the application. The application should display interactive graphics on the screen. We therefore create and store a screen object, representing the output device, and a rasterizer implementation, representing a strategy for drawing graphics to the screen. The rasterizer itself presents a high-level interface to rasterization functionality, implemented by the low-level tools offered by a rasterizer implementation. Again, remember that a rasterizer implementation can be either a software rasterizer implementation, directly manipulating bytes in an off-screen frame buffer, or a hardware rasterizer implementation, calling hardware API functions to instruct the hardware to draw the graphics for us. Therefore, through the rasterizer, rasterizer implementation, and screen, our program has an interface to screen and screen-drawing functionality.



NOTE Theoretically, the screen object could also be created outside of the pipeline. (The following discussion also applies to the rasterizer and rasterizer implementation objects.) There is no technical reason why the screen absolutely must be created within the pipeline constructor. In practice, though, this would make little sense. Consider that the pipeline represents the entire application logic. Creating a screen outside of the pipeline would also mean needing to destroy the screen outside of the pipeline. This would imply some sort of a "higher-level" layer of functionality which creates and destroys objects the pipeline needs in order to function. This would only make sense if the screen object often needed to be used outside of the context of the pipeline, at this "higher-level" layer. Given the current premise that the pipeline is the application, a higher-level layer makes no sense. Therefore, in the current architecture, there is no reason to move management of the screen object outside of the pipeline.

The second responsibility of an `l3d_pipeline` subclass, declaring data, is also intuitive. Since it represents the application, the pipeline subclass must contain all data necessary for maintaining and updating the current state of everything within the virtual world. This might include such things as the current positions and velocities for objects of interest, energy levels for spaceships, the prevailing wind velocity, or anything else being modeled. All of this data is stored within the `l3d_pipeline` subclass in the form of member variables or objects.

The third and final responsibility of an `l3d_pipeline` subclass is to override virtual event functions to respond to events. An `l3d_pipeline` subclass can override any of the following virtual functions declared in `l3d_pipeline`:

```
void key_event(int ch);  //- from dispatcher
void update_event(void); //- from dispatcher
void draw_event(void);   //- from dispatcher
```

The `key_event` function is automatically called whenever a key is pressed in the application window. The function is called with a parameter indicating the ASCII value of the key pressed, thereby allowing the application to respond to the particular key pressed.

The `update_event` function is automatically called whenever the application is allowed to update itself. You can think of your program as being a giant clockwork, with everything happening at each "tick" of the clock. This event function represents one "tick" in your program. At this point you update the internal variables storing the positions of various objects, update velocities, check for collisions, and so on.



TIP The calling frequency of `update_event` is not necessarily guaranteed to be constant. That is to say, the amount of physical time which elapses between successive calls may be slightly different. For accurate physical simulations, where velocities or other physical quantities should be updated based on time, we can store an internal variable recording the value of the system clock the last time that `update_event` was called. We can then compare the current system clock to the value of the variable to determine how much physical time has elapsed, and update the time-dependent quantities accordingly. The particle system program later in this book presents one example of such code.

The `draw_event` function is called whenever the application is allowed to draw its output to the screen. This function typically will be called immediately after `update_event`, but this does not necessarily have to be the case. In other words, the updating of the virtual world and the drawing of the virtual world can be thought of as two separate threads of control, which are usually but not necessarily synchronized.

With this general understanding of a pipeline's structure (creation of screen, storage of variables, and response to events), we can take a closer look at the particular details of the pipeline in the `drawdot` program.

The constructor for the `drawdot` pipeline takes care of the first responsibility of an `l3d_pipeline` subclass: creation of screen, rasterizer implementation, and rasterizer objects. In the constructor, we first ask the screen factory to create a screen and the rasterizer implementation factory to create a rasterizer implementation. We then create a rasterizer which uses the created rasterizer implementation. The member variables `s`, `ri`, and `r` represent the screen, rasterizer implementation, and rasterizer, respectively.

The constructor also takes care of the second responsibility of an `l3d_pipeline` subclass: management of data representing our virtual world. In our case, our virtual world consists of a single pixel (a humble start). The following member variables are declared and initialized to keep track of the dot's status: `color`, `x`, `y`, `dx`, and `dy`. Variables `x`, `y`, `dx`, and `dy` represent the dot's current horizontal and vertical positions and velocities, and are all initialized to zero. The variable `color` represents the dot's current color, and is specified as follows. First, we logically define the maximum red, green, and blue values to be 255. Then, we specify a color of (0, 255, 128), which means a red intensity of 0, a green intensity of 255, and a blue intensity of 128, all being measured in relation to the logical maximum of 255 which we just set. Finally, we convert this RGB color to a "native" color appropriate for the current screen's color depth and color model. The conversion is done via an object of type `l3d_screen_info`, which encapsulates the complicated color calculation discussed earlier. The color conversion function is called `ext_to_native`, as it changes a color from an "external" RGB format into a format "native" to the XImage.

The `drawdot` pipeline then overrides the `key_event`, `update_event`, and `draw_event` methods to respond to events. This fulfills the third and final responsibility of an `l3d_pipeline` subclass, responding to events.

The `key_event` for the `drawdot` pipeline checks to see if any one of the directional keys was pressed, and updates the `dx` and `dy` variables, representing the horizontal and vertical velocities, accordingly.

The `update_event` for the `drawdot` pipeline adds the velocities to the positional variables, and makes sure the position stays within the bounds of the screen. In other words, `x += dx` and `y += dy`.

The `draw_event` for the `drawdot` pipeline first calls the `draw_point` routine of the rasterizer, which then forwards the request to the rasterizer implementation to draw a pixel at a particular point in a particular color. Remember that the drawing occurs off-screen (double buffering). The pixel color must be specified in "native" format for the current color depth and color model. We already computed and stored this color earlier by using the function `l3d_screen_info::ext_to_native`. After plotting the point, we call `blit_screen` to cause the off-screen graphics to be copied to the screen.

Let us summarize the main idea behind the pipeline. A pipeline represents the main functionality of an application and is subclassed from `l3d_pipeline`. An `l3d_pipeline` subclass has three responsibilities: creating screen-access objects, declaring world data, and responding to events. Creating screen-access objects (screen, rasterizer implementation, and rasterizer) allows access to the screen and screen-drawing functions. Declaring world data allows the program to keep track of the state of all objects in the virtual world. Responding to events is how the pipeline responds to input (through `key_event`), updates the virtual world (through `update_event`), and draws to the screen (through `draw_event`, using the previously created screen-access objects).

The pipeline does not need to worry about how or when events occur; it merely responds to them. The pipeline's virtual event functions are thus called from an outside source. This outside source is the *dispatcher*.

Step 3. Create a Dispatcher

The third step in writing an l3d application is to create an *event dispatcher* object. The event dispatcher serves as an interface between an event source and an event receiver. The event receiver in our case is the pipeline. The event source is a window created under a specific event-driven windowing system. The role of the dispatcher is to receive events from the system-specific window, and to call the appropriate pipeline functions to allow the pipeline to respond to the events.

The whole idea is to isolate the pipeline (i.e., your application logic) from the details of the underlying event-generating mechanism. This way, the pipeline's logic can focus exclusively on application-specific responses to events, without needing to know exactly how the windowing system generates and transmits events. The dispatcher handles all the messy details of event capturing and translates this into a clean, simple, virtual function call to the pipeline. This allows your pipeline to work on a variety of platforms, with a variety of event-generating mechanisms.

The event dispatcher must be created using the factory chosen in step 1. This is because the dispatcher represents system-specific code, and should thus be created through an abstract factory.

Step 4. Create a Pipeline

The fourth step in creating an l3d application is to create your pipeline object. This step is easy. Having already declared and defined an `l3d_pipeline` subclass, which fulfills the three pipeline responsibilities (creating screen-access objects, declaring world data, and overriding event-handling functions), we simply create the pipeline directly with the C++ `new` operator. This, in turn, invokes the pipeline's constructor, which creates the screen, rasterizer implementation, and rasterizer objects.

At this point, the application is ready to respond to events. We just need to pump events to the pipeline in order to allow it to respond to input, update itself internally, and draw to the screen. To start the entire event process, we start the dispatcher.

Step 5. Start the Dispatcher

The fifth step in writing an l3d application is to start the dispatcher. We must do three things:

1. Assign a pipeline to the dispatcher.
2. Assign an event source to the dispatcher.
3. Call `start`.

A moment's reflection makes it clear why the two assignments are necessary. The dispatcher takes events from the event source, interprets them minimally, and calls the appropriate pipeline virtual event function to allow the pipeline to respond. The `pipeline` member of the dispatcher object is set to the pipeline we just created. The `event_source` member of the dispatcher object is set to the screen object created in the pipeline's constructor—in other words, the screen (in our case, the X window) is the source of events. With these two member variables set, the dispatcher can then begin to extract events from `event_source` and pass them on to `pipeline`—a process set in motion by calling `start`.

Summary of Fundamental l3d Concepts

The five-step process presented above is typical of l3d programs. First, you choose the proper factories to configure the program to its environment. Then, you declare a pipeline representing your application and its data. You create an instance of the pipeline, which in turn creates screen, rasterizer implementation, and rasterizer objects. You "plug" this pipeline into an event dispatcher. Your application pipeline responds to events from the dispatcher by filling in the blanks left by the virtual functions `key_event`, `update_event`, and `draw_event`. The application pipeline draws to the screen by using the screen, rasterizer implementation, and rasterizer objects it created within its constructor. This forms a complete, interactive, event-driven, hardware-independent graphics program.

Overview of l3d Classes

We are now ready to look at the specific l3d classes. A much more detailed development of the following l3d classes appears in the introductory companion book *Linux 3D Graphics Programming*. The descriptions below are more of a reference, but suffice for a high-level understanding of the structure of the library classes.



TIP Although the descriptions of the l3d classes presented here are not as detailed as the full-blown development in the companion book, all of the sample programs and library code from the introductory companion book are also included on the CD-ROM. Therefore, you can study the provided code to understand more precisely the library concepts summarized in the following sections.

Applications and Events

Applications written with l3d are event-driven. The classes described in the following sections deal with applications, events, and event handling.

An Event-driven Graphics Application: l3d_pipeline

The class `l3d_pipeline`, in file `pipeline.cc`, is an abstract class representing your application. As we saw earlier, you subclass from `l3d_pipeline` to create your application, and must fulfill three responsibilities: creating the screen-access objects, declaring world data, and responding to events.

Event Generation: l3d_event_source

Class `l3d_event_source`, in file `ev_src.h`, is an empty class whose sole purpose is to indicate that any class derived from this class is capable of serving as an event source for an `l3d_dispatcher` object. A class will inherit, possibly multiply inherit, from `l3d_event_source` if it is usable as a source of events. Class `l3d_screen` inherits from `l3d_event_source`.

The Event Dispatching Loop: l3d_dispatcher

The class `l3d_dispatcher`, in file `dispatch.h`, is an abstract class representing an event-dispatching mechanism. It extracts events from an underlying event-generating mechanism and translates these into virtual function calls on a pipeline, thereby allowing the pipeline to respond to events. Class `l3d_dispatcher` is a recognition of the general concept of an event dispatching mechanism, which can be subclassed to provide support for event dispatching on a variety of platforms other than Linux and X (e.g., Linux and Mesa, or Microsoft Windows and a Windows window). This is an example of the *strategy design pattern*, where an algorithm (in this case, the event loop) is turned into a class of its own.

Figure 1-8 illustrates the relationship among the abstract classes `l3d_pipeline`, `l3d_dispatcher`, and `l3d_event_source`.

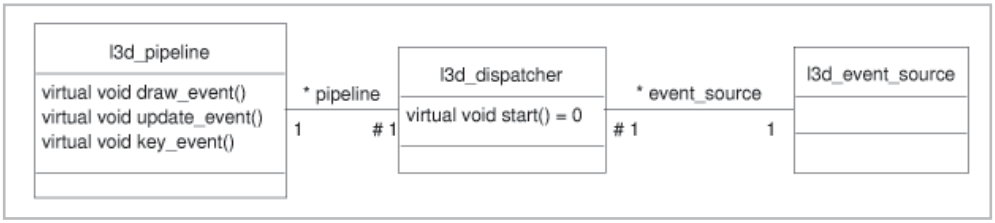


Figure 1-8: Class diagram for event-related classes.

The class `l3d_dispatcher_x11`, subclassed from `l3d_dispatcher`, represents a dispatcher specific to the X Window System. The source file is `dis_x11.cc`. The class `l3d_dispatcher_mesa` (file `dis_mesa.cc`) represents a dispatcher working within the GLUT framework provided for OpenGL and Mesa.

2D Graphics

Current mainstream display hardware for personal computers is for all practical purposes flat and two-dimensional. The classes described in the following section deal with accessing 2D screen and drawing 2D raster graphics.

Control of the Screen: l3d_screen

The classes `l3d_screen` (file `screen.cc`) and `l3d_screen_info` (file `scrinfo.h`) work closely together to control and provide access to the display hardware.

The class `l3d_screen` is an abstract interface to a display device. A screen is responsible for the creation, setup, and display of the data which has been plotted to the screen. However, the screen is not responsible for doing the plotting itself. In other words, the screen is a "dumb" display device. It can show itself, initialize a color palette, and so forth, but it does not know how to draw polygons, lines, dots, or anything else. The screen is simply a passive display device which other objects can use as a target to draw into. In particular, the `l3d_rasterizer` class handles plotting (rasterization) tasks, manipulating data within the screen. Class `l3d_screen` is a recognition of the general concept of a screen as a generalized output device—not just an X Window

with an XImage. The application program uses a screen through the abstract `l3d_screen` interface, and is therefore not tied to any particular display device.

The class `l3d_screen_x11`, subclassed from `l3d_screen`, represents a screen under the X Window System, in the form of a window using an XImage for graphics display. The source file is `sc_x11.cc`. The class `l3d_screen_mesa` (file `sc_mesa.cc`) represents a screen created using GLUT and Mesa.

Relevant Screen Attributes: l3d_screen_info

The class `l3d_screen_info` (file `scrinfo.h`) is an abstract interface to screen information. We define screen information as follows: any information which an external class needs to know about a screen object in order to be able to work with that screen. Class `l3d_screen_info` encapsulates the vital statistics about a screen and makes them available through a clean, easy interface.

In particular, when dealing with colors and XImages, the bytes making up a pixel and their interpretation depend on the color depth and color model of the underlying X server. Class `l3d_screen_info` is a recognition of the more general concept that external users of a screen need to access such "screen information" in order to do anything useful with the screen.

Such screen information includes:

- Maximum ranges for the specification of red, green, and blue (RGB) values
- A conversion function to convert colors specified in RGB values into native color format (i.e., the exact bit format) needed by the underlying screen
- Number of bytes per pixel
- A pointer to the memory of the off-screen buffer (not applicable for hardware-accelerated display devices, where such access is usually not possible)
- A method for setting internal color state (needed for OpenGL)
- Methods for computing lighting and fog tables for gradual fading of colors, needed for light and fog effects
- Methods for applying lighting and fog tables to a particular color to obtain the resulting lit or fogged color



NOTE It might appear tempting to merge `l3d_screen_info` into the `l3d_screen` class itself. After all, isn't screen information part of the screen itself? The problem appears when we try to subclass the screen. Screen information can be handled fundamentally in two different ways: as RGB (TrueColor) or as indexed color. Similarly, screens themselves come in a variety of sorts: X11 screens, Mesa screens, Windows screens, DOS screens. If screen information and the screen were merged into one class, we would have the unpleasant situation of having several sorts of each screen: an X11 RGB screen, an X11 indexed screen, a Mesa RGB screen, a Mesa indexed screen, a Windows RGB screen, a Windows indexed screen, a DOS RGB screen, a DOS indexed screen. Extending the class hierarchy with a new information type or a new screen type becomes a major headache. This situation is sometimes called a nested generalization and indicates that the class should be split into two. For this reason, we keep the screen information separate, in its own `l3d_screen_info` class hierarchy. The `l3d_screen` is also separate, in its own `l3d_screen` class hierarchy. We can then, at run time, mix and match screen information types and screen types freely, without the

multiplicative explosion of classes which would have resulted had we combined `l3d_screen_info` and `l3d_screen` into one class.

The class `l3d_screen_info_rgb`, subclassed from `l3d_screen_info`, represents screen information for X TrueColor visuals or other color models based upon a direct specification of red, green, and blue pixel values. The source file is `si_rgb.cc`. The class `l3d_screen_info_indexed` (file `si_idx.cc`) represents screen information for X PseudoColor visuals or other color models based upon an indexed or paletted color model.

Figure 1-9 illustrates the class diagram for the screen-related classes.

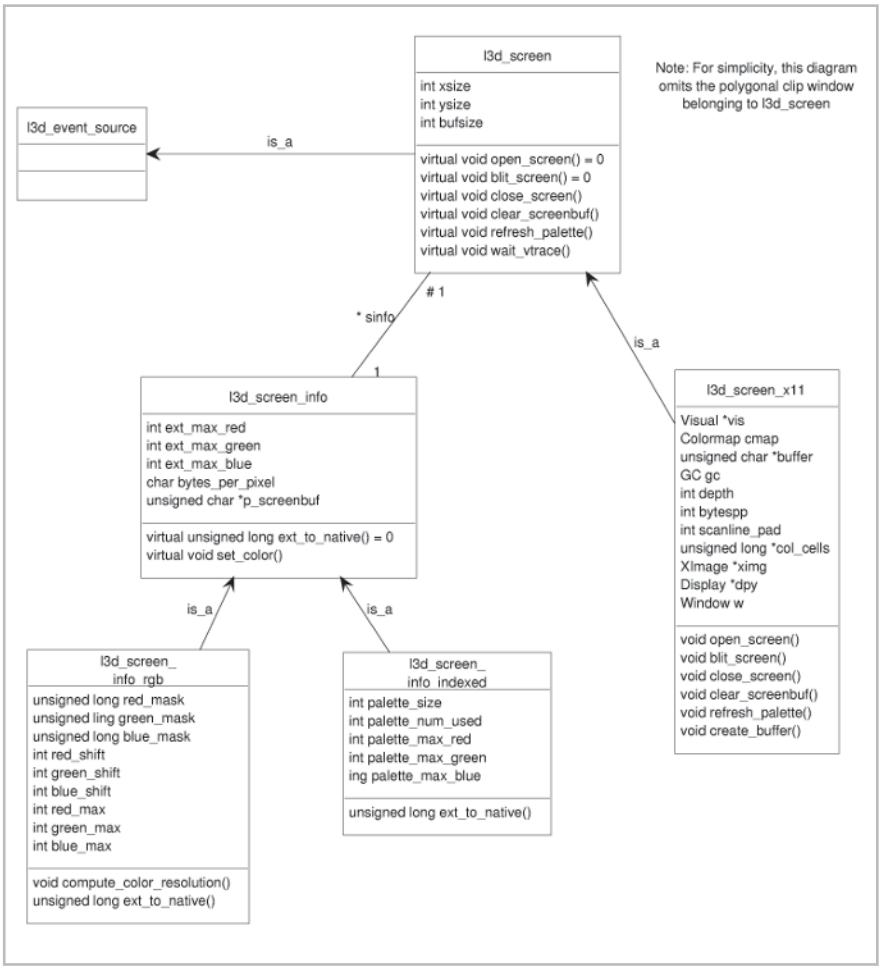


Figure 1-9: Screen-related classes.

The Rasterizer: l3d_rasterizer_2d

The class `l3d_rasterizer_2d` (file `rasteriz.cc`) represents a 2D, double-buffered rasterizer—a subsystem capable of drawing 2D geometric primitives on a double-buffered raster output device. The class does not do any of the rasterization work itself, but instead relies on a rasterizer implementation (covered in the next section) to provide a "toolkit" of functions that the rasterizer calls. Class `l3d_rasterizer_2d` is a recognition of the general concept of a rasterizer interface, which allows access to abstract rasterization concepts. The rasterizer interface allows drawing not only of individual pixels, but also of groups of pixels forming geometric primitives, such as lines and polygons.



NOTE Separating the rasterizer and the rasterizer implementation is an example of the generally applicable *bridge design pattern*, where an abstraction and its implementation are two separate classes. This allows for the class hierarchies for the abstraction and the implementation thereof to vary and be refined independently, avoiding the multiplicative explosion of classes discussed earlier (nested generalizations). This technique is also referred to as using a *handle*.

The class `l3d_rasterizer_2d_imp` (file `rasteriz.cc`) represents an implementation of the tools necessary for a 2D rasterizer to do its work, and is called a rasterizer implementation. A rasterizer implementation is an abstract class, and is subclassed to provide concrete rasterizer implementations for specific software or hardware platforms. While the related class `l3d_rasterizer_2d` represents an interface to 2D rasterizer functionality, the class `l3d_rasterizer_2d_imp` represents an interface to an implementation of 2D rasterizer functionality. This rasterizer implementation can (in subclasses) be based upon either hand-coded software algorithms, or an API to hardware-accelerated rasterization functions.

The class `l3d_rasterizer_2d_sw_imp`, derived from class `l3d_rasterizer_2d_imp`, represents a rasterizer implementation implemented in software (as opposed to hardware). The source file is `ras_sw.cc`. The calculation of the bytes corresponding to a pixel and setting these bytes in order to display geometric primitives are all operations typical of software rasterization implementations, but not necessarily of hardware rasterizer implementations. This is the reason that such byte and pixel calculations take place only in the concrete `l3d_rasterizer_2d_sw_imp` subclass, which represents a software rasterizer implementation, and not in the abstract ancestor class `l3d_rasterizer_2d_imp`, which represents a more abstract concept of a software or hardware rasterizer implementation.

Class `l3d_rasterizer_2d_mesa_imp` (file `ras_mesa.cc`) represents a rasterizer implementation implemented using OpenGL/Mesa calls. Through hardware support in the Mesa library, or through the Direct Rendering Interface, this rasterizer implementation can take advantage of hardware acceleration.

Figure 1-10 illustrates the class diagram for rasterization-related classes.

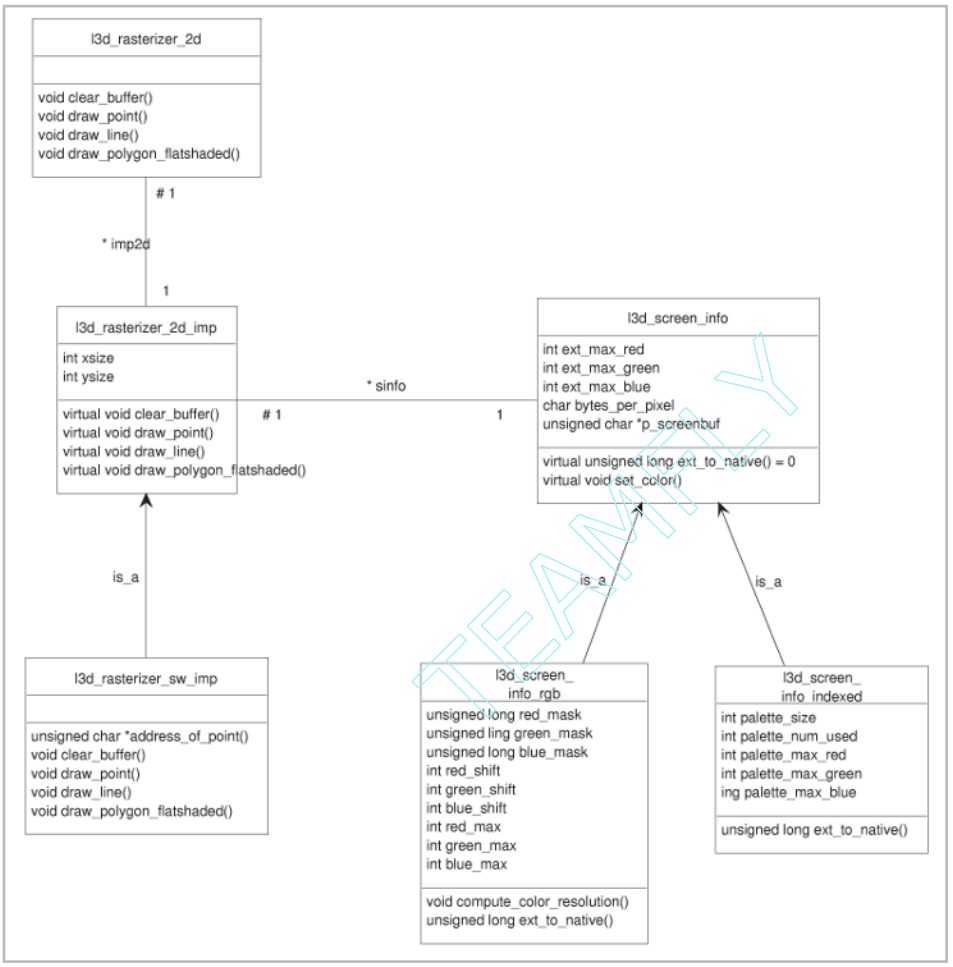


Figure 1-10: Rasterization-related classes.

Concrete Factory Management

The class `l3d_factory_manager` (file `factorys.cc`) represents the factory manager which contains all factories which any class might need in the entire program. Whenever an object requests object creation through a "factory," this factory is to be found in the central factory manager.

Global variable `factory_manager_v_0_1` is a global instance of the factory manager. An application chooses its factories at startup through the global variable `factory_manager_v_0_1`. Thus, the factory manager class is a singleton class, which is instantiated only once in the entire program.

Notice that this is the only l3d class which directly manipulates the concrete factories by their class name, and which includes the header files for the concrete factories. All other users of the factories always go through an abstract factory interface, and do not include the header files for the concrete factories. This is the whole point of the abstract factory: to completely isolate the

Team-Fly®

application from the underlying concrete types to allow for future extension of code without any change of original code.

Specifying Geometry and Behavior

The classes described in the following sections allow l3d programs to specify objects in 3D and to manipulate them.

Points

Specifying points is done through two classes: `l3d_point` (file `point.cc`) and `l3d_coordinate` (file `coord.h`).

The class `l3d_point` represents a set of four numbers, of the form (x, y, z, w). These numbers represent a location in space in terms of displacements along coordinate axes.

The class `l3d_coordinate` represents a transformable coordinate location in space. It stores two variables of type `l3d_point`, named `original` and `transformed`. The transformed coordinate is always initialized to equal the original coordinate at the beginning of each frame (method `reset`), after which it may be transformed by zero or more transformations (method `transform`). After any number of transformations, the application may choose to save the current location in one of the intermediate storage locations (member `transformed_intermediates`), so that the old location may be referenced later. In the end, the transformed coordinate must represent a point in 2D pixel space, since ultimately the points must be used to plot images on-screen. The whole idea is that an `l3d_coordinate` is more than just a single location in space: it is an original location, a number of optional intermediate locations, and a final location.

The most important thing to remember about the `original` and `transformed` members is that the transformed coordinate is the one that is finally used to display the pixels on the screen. An application must at some point put the final pixel coordinates in the `transformed` member variable.

Lists

Specifying more complex geometry (as we see in the coming section on polygons) requires us to store lists of vertices. For this, we use a combination of two data structures: `l3d_list` and `l3d_two_part_list` (both in file `list.h`). Class `l3d_list` is a dynamically growable list that is like an array, but which automatically expands and allocates more space as necessary. The elements are accessed as normal through the array index operator []. Class `l3d_two_part_list` is an extension of `l3d_list` and partitions the list of items into two parts: a fixed part and a varying part. The fixed part is fixed in size and never changes; the varying part is based on some dynamic calculation and changes often in size. However—and this is the whole point of the `l3d_two_part_list`—both the fixed and the varying parts are accessed identically. If we stored the fixed and varying parts in two separate lists, any references to list items would need to specify if the item comes out of the fixed or the varying list, which makes for rather inconvenient code. With the `l3d_two_part_list`, external users don't need to worry about whether the referenced item is fixed or varying. This uniformity of access is primarily of use in *polygon*

clipping. Clipping can create temporary (varying) vertices which must be accessible exactly like normal (fixed) vertices, but which may vary wildly in number from frame to frame.

Using these list classes is easy. They are declared as template classes, so that they work with any data type. Let's first begin with the `l3d_list` class.

To use `l3d_list`, do the following:

1. Declare an instance of the list, providing the data type as the template parameter and passing the initial size of the list to the constructor. The initial size must be greater than zero.
2. Call `next_index` before storing any new element into the list, to obtain the next free index within the list. This also causes the list to grow if you exceed the current size of the list.
3. Store and access elements in the list using the array index operator []. Never attempt to access an element you have not already allocated by having called `next_index`.
4. Query the number of items via the `num_items` member. Valid indices range from 0 to `num_items` –1.
5. Effectively empty or reduce the size of the list by setting `num_items` to zero or a smaller number. Never set `num_items` to a larger number; use `next_index` to increase the size of the list.
6. Copy the list by assigning it to another `l3d_list` object. This makes a full copy of the list and its contents. The two copies are completely independent of one another and contain no pointers to common objects (unless the contents of the list themselves are pointers, in which case the two lists contain separate copies of pointer objects, which point to the same locations in memory).



NOTE The creation of the list elements occurs through an abstract factory; the copying of list elements, through a possibly virtual assignment operator; the access of list elements, through an abstract class pointer in the overloaded array index operator []. This means that by providing a new factory (via the `l3d_list` constructor) and overriding a virtual assignment operator =, an existing `l3d_list` in already existing code can be extended after the fact to work with new subclasses, and the original code is still guaranteed to work because the internal access to the list elements is done through an abstract class pointer. A detailed example of this technique appears in Chapter 2, when we extend the existing polygon class to store texture coordinate information with each vertex, without needing to change any of the already existing code. By default, an abstract factory is automatically created and destroyed if you do not provide one; the flexibility is there if you need it, but does not burden you if you do not.

The other list class is the `l3d_two_part_list`. Again, a two-part list consists of a fixed part, whose size never changes, and a varying part, whose size may change greatly over time. The list elements in both the fixed part and the varying part are accessed identically (with the array operator []). To use an `l3d_two_part_list`, do the following:

1. Declare an instance of the list, providing the data type as the template parameter and passing the initial fixed size of the list to the constructor. The initial size must be greater than zero.
2. Store and access fixed items in the list by using the array index operator []. Valid indices for fixed items range from 0 to `num_fixed_items` –1. Do not store fixed items outside of this range. Do not change the `num_fixed_items` variable.

3. Call `next_varying_index` before storing any element in the varying part of the list, to obtain the next free index within the varying part. This also causes the varying part of the list to grow if you exceed the current size of the list.
4. Store and access elements into the varying part of the list using the array index operator []. Never attempt to access an element in the varying part which you have not already allocated by having called `next_varying_index`.
5. Query the number of varying items via the `num_varying_items` member. Valid indices for varying items range from `num_fixed_items` to `num_fixed_items + num_varying_items` – 1.
6. Effectively empty or reduce the size of the varying part of the list by setting `num_varying_items` to zero or a smaller number. Never set `num_varying_items` to a larger number; use `next_varying_index` to increase the size of the varying part of the list.
7. Copy the list by assigning it to another `l3d_two_part_list` object. This makes a full copy of the fixed and varying parts of the list and its contents; the two copies are completely independent of one another and contain no pointers to common objects (unless, again, the contents are themselves pointers).

Notice that access to both fixed and varying items is done uniformly through the array access operator [].



NOTE The earlier comments about factory creation, virtual assignment, and polymorphic access all apply to `l3d_two_part_list` as well. This is because `l3d_two_part_list` internally uses an `l3d_list` to store the items.

We use two-part lists for storing vertices (i.e., objects of type `l3d_coordinate`) which are later used to define polygons. A typical creation of a vertex list looks like the following:

```
vlist = new l3d_two_part_list<l3d_coordinate> ( 4 );
(*vlist)[0].original.X_ = float_to_l3d_real(100.0);
(*vlist)[0].original.Y_ = float_to_l3d_real(100.0);

(*vlist)[1].original.X_ = float_to_l3d_real(200.0);
(*vlist)[1].original.Y_ = float_to_l3d_real(150.0);

(*vlist)[2].original.X_ = float_to_l3d_real(150.0);
(*vlist)[2].original.Y_ = float_to_l3d_real(200.0);

(*vlist)[3].original.X_ = float_to_l3d_real( 50.0);
(*vlist)[3].original.Y_ = float_to_l3d_real(120.0);
```

First, we create an object of type `l3d_two_part_list` with a data type of `l3d_coordinate`. The constructor parameter 4 indicates the size of the fixed part of the two-part list. We then initialize the first four elements, with indices 0 through 3, simply by using the array access operator []. Each element of the list is an object of type `l3d_coordinate`, which we can manipulate directly. In the code snippet above, we initialize the `original` member of the `l3d_coordinate`, which is what an application program does to initially define a vertex list. The varying part of the two-part list is not used above, but is used during polygon clipping operations.

Vectors and Matrices

In addition to specifying points in 3D space, we also need to specify vectors, which are displacements, and matrices, which represent transformations. Class `l3d_vector` (file `vector.cc`) represents vectors; class `l3d_matrix` (file `matrix.cc`) represents matrices.

The l3d source code defines various operators allowing the appropriate mathematical operations among scalars, points, vectors, and matrices. For instance, we can multiply two matrices together, multiply a matrix and a point, add a point and a vector, and so forth. Note that the pipe character (the vertical bar "|") is used to specify matrix multiplication.

Polygons

The l3d classes make a distinction between the geometry and the appearance of a polygon. The base classes `l3d_polygon_2d` and `l3d_polygon_3d` describe the geometry of a polygon in 2D or 3D, respectively. Some typical geometric operations on polygons are clipping, transformation, or surface normal computation. We subclass the geometric polygon classes to obtain drawable polygon classes. For instance, `l3d_polygon_3d_flatshaded` inherits the attributes and methods used for describing the geometry of a 3D polygon, and adds new attributes and methods necessary to draw the polygon on-screen.

A polygon requires two lists for its definition: an external vertex list, and an internal list of indices into the external vertex list. The external vertex list can be shared among several polygons, as is the case when we combine several polygons to make an object. The internal list of indices specifies the polygon in terms of the external vertices, specified in clockwise order when looking at the front of the polygon (the left-handed rule). During an analytic clipping operation, new vertices can be created; this requires addition of new vertices to the external list, and temporary redefinition of the internal vertex index list to use the new clipped vertices.

The source code for the polygon classes is located on the companion CD-ROM in the directory `$L3D/source/app/lib/geom/polygon`.

Objects

Class `l3d_object` (file `object3d.h`) represents a 3D polygonal object. It declares a list of vertices and a list of polygons. The vertices are shared among all polygons. Each polygon is defined in terms of a list of indices into the shared vertex list of the object.

The class `l3d_object` stores a list of matrices representing transformations on the object. The method `reset` sets all vertices of the polygon back to their original positions in object coordinates; the method `transform` then applies a transformation to the vertices, so that the vertices may be moved into world space, camera space, and finally, screen space. The method `camera_transform` is identical to the method `transform`, except that it calls a user-definable routine after the transformation. This allows us to save the camera-space coordinates, which we need for texture mapping (see Chapter 2). Although the perspective projection could be handled with a matrix, the class `l3d_object` implements a specialized routine `apply_perspective_projection,` which projects the vertices of the polygon from 3D into 2D.

Polygon culling is done with the help of the linked list `nonculled_polygon_nodes`. Initially, at the beginning of each frame, the list contains all polygons in the object. Through various culling processes, we can remove polygons from the list if we determine them to be not visible

for the current viewpoint (see Chapter 4). Then, for each frame, we only need to draw the polygons that remain in the list.

Each object can update itself. This allows construction of 3D worlds in a modular fashion, with a number of "intelligent" objects capable of acting on their own. An object can update itself by changing its geometry (i.e., altering the vertices in the vertex list), or by applying a transformation matrix to its geometry. Such updates can take place either through subclassing or through plug-ins. Wíth subclassing, a new object subclass overrides the virtual method `update`, which updates the object as needed. With a plug-in, we write a dyamic shared library file which is loaded at run time and which then updates the object.

Worlds

Just as groups of 3D polygons form a 3D object, groups of 3D objects form a *world*. Class `l3d_world` (file `world.h`) is the class we use for storing worlds. A world is a displayable, interactive 3D environment. It contains 3D objects to populate the world, a camera to see the world, a rasterizer to plot pixels, and a screen to display the output.

A simple world as implemented by `l3d_world` is really little more than a simple list of objects displayed with the painter's algorithm. Later, we subclass from `l3d_world` to perform more advanced visibility operations on our 3D objects, leading to a graph-like structure of objects connected via portals. See Chapter 5 for details.

For interaction with the external events (such as the user pressing a key on the keyboard), the world class relies on a corresponding pipeline class `l3d_pipeline_world` (file `pi_wor.h`). The world pipeline class merely forwards events on to the world to be handled.

Fixed- and Floating-Point Math

In 3D graphics, we generally need to be able to store and manipulate fractional values. For this, we introduce the type `l3d_real`. The purpose of the `l3d_real` type is to allow all computations using real values to be performed using either floating-point or fixed-point arithmetic. *Floating-point* arithmetic is done by using the C type `float`, and for good real-time performance requires a hardware floating-point unit. *Fixed-point* arithmetic requires no floating-point unit in hardware, and simulates fractional values with integers by multiplying the fractional value so that the fractional part moves into the integer part.

By using `l3d_real` any time we want a real value, we can configure our code at compile time to use either fixed- or floating-point math. There are four functions you need to know for using variables of type `l3d_real`:

- Type conversions. A variable of type `l3d_real` can be converted to and from an integer or floating-point value through the macros `l3d_real_to_float`, `l3d_real_to_int`, `float_to_l3d_real`, and `int_to_l3d_real`. Furthermore, a variable of type `l3d_real` can be converted to and from a fixed-point number through the macros `l3d_real_to_fixed` and `fixed_to_l3d_real`.
- Addition and subtraction. Variables of type `l3d_real` may be added to and subtracted from other variables of type `l3d_real` with the normal plus (+) and minus (–) operators. Addition and subtraction with variables of any other type must first use one of the type conversion macros to convert the variable of the other type to `l3d_real`.
- Multiplication and division. Variables of type `l3d_real` can be multiplied or divided by variables of type `l3d_real` or of type `int` only. Such multiplications or divisions must use one of the following macros, all of which return a type of `l3d_real`: `l3d_mulrr` (multiplies an `l3d_real` by an `l3d_real`), `l3d_mulri` (multiplies an `l3d_real` by an `int`), `l3d_divrr` (divides an `l3d_real` by an `l3d_real`), and `l3d_divri` (divides an `l3d_real` by an `int`).
- Other functions. The `iceil` function returns the integer ceiling of (i.e., the smallest integer larger than) an `l3d_real`. `ifloor` returns the integer floor of (the largest integer smaller than) an `l3d_real`. The function `l3d_sqrt` returns an `l3d_real` representing the square root of the given `l3d_real` argument.



NOTE The reasons for not making `l3d_real` a class are discussed in the Appendix of the introductory companion book, *Linux 3D Graphics Programming*.

If we choose to use floating-point math (at compile time via a `#define`), then the `l3d_real` macros do nothing other than call the ordinary cast, multiplication, or division operators. If we choose to use fixed-point math (again, at compile time via a `#define`), the `l3d_real` macros are defined quite differently, applying the somewhat arcane rules for fixed-point math to simulate real-valued computations using purely integer arithmetic.

An unfortunate consequence of using the `l3d_real` macros for multiplication and division is that this leads to somewhat awkward code. For instance, the following computation with floating-point values:

```
x = a * ( ( b + c ) / d )
```

would need to be expressed with `l3d_real` types as:

```
x = l3d_mulrr( a , l3d_divrr( b + c , d ) )
```

The multiplication and division must therefore be realized as nested macro invocations. Notice, however, that no macro invocation is necessary for the addition of `b` and `c`, since addition and subtraction of `l3d_real` types can be done with the normal addition and subtraction operators.

Summary of l3d Classes

Figure 1-11 illustrates the major l3d classes covered up to this point. Study the previously presented `drawdot` program to keep a practical perspective on the use of the l3d classes, and remember the five-step process for creating l3d applications. Also, remember that the CD-ROM contains complete source code listings for all of these classes, and the introductory companion book *Linux 3D Graphics Programming* develops the classes in detail, with several example programs.

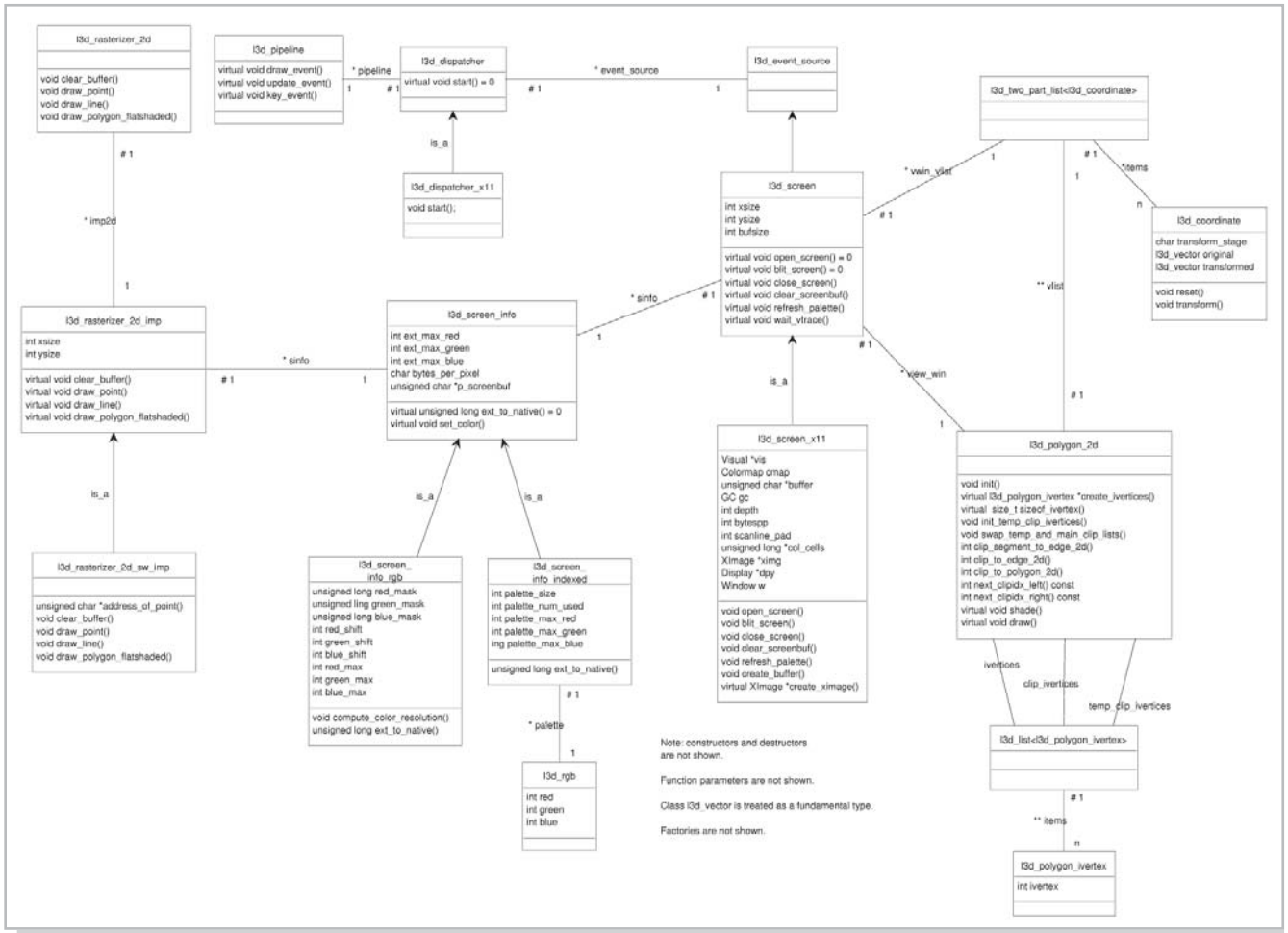


Figure 1-11: Class diagram for the most important l3d classes covered so far. For space reasons, not all classes are displayed; the naming convention makes it obvious where the other classes would appear in the diagram. For instance, `l3d_screen_mesa` *is a descendant of* `l3d_screen`*, as is clear from the naming convention.*



TIP See the source code for the sample program `fltsim` on the CD-ROM (taken from the introductory companion book *Linux 3D Graphics Programming*) for a typical example of creating an animated world populated with polygonal objects and a user-controllable camera.

Linux Programming Tools

This book assumes you are already fairly comfortable programming on a Linux system. I won't spend any time explaining how to change directories, start the X Window System, set environment variables, or bring up a command shell. The following list, while not complete, mentions a number of important Linux programming tools with which you should be familiar. You don't have to be an expert with all of these tools, but it is important to know of their existence, to be able to perform basic operations with these tools, and to know how to find out more about these tools.

- man: The online manual. Use this command to find information about other commands.
- info: The online info documentation, often used for GNU projects or for very large documentation. Use this command to find information about other programs not documented with "man."
- Emacs: The most powerful editor around. We use an Emacs module in Chapter 2 to perform symbolic algebra. You should be comfortable editing files and executing commands in Emacs in order to be able to carry out the symbolic algebra exercises in Chapter 2.
- gcc: The GNU C/C++ compiler.
- gdb: The GNU debugger. It is text-based, but several graphical front ends exist.
- make: The project-compiling utility. You should have a basic understanding of what Makefiles are and how they are interpreted, since practically every non-trivial Linux project uses "make" to direct the actual compilation of the source code files.
- binutils: A collective term for the GNU binary utilities. Use these tools to find out information about system libraries and object files. This can help in finding out which library file contains a particular function with which you need to link your program.
- find: A powerful file-finding program. Use "find" to look for files (such as C++ header files) containing a particular string or matching some criteria.

Linux 3D Modeling

The introductory companion book *Linux 3D Graphics Programming* covered basic usage of the Blender 3D modeling package for creating 3D models. We continue the coverage in this book. Blender is zero-cost software; a fully functional version of Blender, with no artificially imposed restrictions, is included on the CD-ROM.



NOTE The term "zero-cost software" means that you can use the software without having to pay for the software. The difference between "zero-cost software" and "free software" is that free software comes with source code, and can be freely modified. "Free software" refers to freedom, not price. Free software by its open nature is usually zero-cost software, but the reverse is not true. The GNU/Linux operating system is free software, as is the source code included with this book. See `http://www.gnu.org` for a full definition of "free software." Remember, your freedom as a professional programmer is literally at stake here.

The rest of this section briefly goes over the most important features of Blender as discussed in the introductory companion book.

Blender Interface and Commands

The Blender interface consists of a number of non-overlapping windows. Each window has a header and a type. You may choose to display the window header either at the top or the bottom of a window by right-clicking on the header. Change the type of the window by clicking and dragging the button on the left side of the window header. You can change the type of a window at any time. The types of windows are:

- InfoWindow: Contains global settings for the entire program, such as directory names.
- 3DWindow: Displays 3D objects and allows creation, positioning, and editing of these objects.
- ButtonsWindow: Edits properties of the current selection in the 3D window, or of global editing or rendering parameters.
- FileWindow: Selects files for loading and saving.
- OopsWindow: Displays the internal object-oriented programming structure (OOPS) of the currently selected scene in a tree format. Editing is not allowed in this window.
- SoundWindow: Allows integration of sounds into interactive 3D environments created with Blender. In Blender 2.0 beta, this window type is not yet fully functional.
- IpoWindow: Displays and edits interpolators (IPOs), which are used for interpolating various object parameters over time. Blender has a very flexible interpolation system where many parameters can be interpolated and thus animated over time; interpolators can animate objects, vertices, textures, lamps, or sequences.
- SequenceWindow: Combines and mixes among various animation sequences. This window type is essentially a post-production video studio, where each animation sequence is represented as a horizontal strip with the length indicating the duration. Strips can be blended or combined with one another, allowing, for instance, smooth fades between different parts of a 3D animation. Another exciting possibility is the combination of Blender rendered animations with existing video footage from another source, such as a video camera.
- TextWindow: Edits a text file. Used primarily for creating script files for the built-in Python extension language.
- ImageWindow: Displays an image. Useful for previewing texture image files; also used for assigning specific texture coordinates to specific faces of a mesh.

- ImageSelectWindow: Selects an image file for loading or saving, displaying a thumbnail preview of each image file for easy differentiation.

The majority of the modeling work in Blender is done in the 3DWindow, where geometry is interactively created in 3D, and the ButtonsWindow, where various properties and parameters can be set. See Chapters 3 and 6 for further discussion of Blender and some of the other window types.

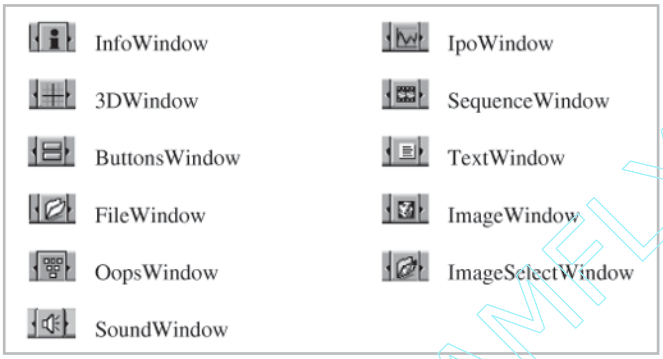


Figure 1-12: Window types in Blender and their appearance in the window header.

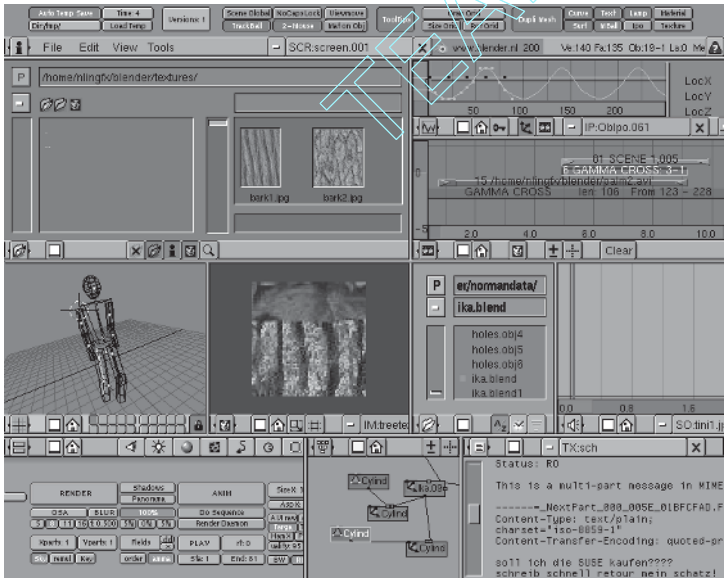


Figure 1-13: All Blender window types open simultaneously.

New windows in the Blender user interface are created by splitting existing windows. A border separates each window header from the window contents. Middle-click the border to split the window. Right-click a border to join two adjacent windows.

A window can receive keyboard input only when the mouse cursor is positioned over that window. As mentioned earlier, the actual 3D modeling work in Blender is done in the 3DWindow. During modeling, the most important Blender commands can be executed by an appropriate single-keystroke command within the 3DWindow. The following list summarizes the most important single-keystroke commands in the 3DWindow:

TEAMFLY

- Ctrl+Up Arrow: Toggle maximization of window.
- 1, 3, 7 on the numeric keypad: Display front, side, or top view.
- 5 on the numeric keypad: Toggle between perspective and orthogonal viewing.
- Tab: Toggle EditMode. In EditMode, you edit and move individual vertices within one single object; outside of EditMode, you position and move complete objects.
- Shift+middle-click+drag: Translate viewpoint.
- Ctrl+middle-click+drag: Zoom view.
- Middle-click+drag: Rotate view.
- Left-click: Position 3D cursor.
- Right-click: Select object (outside of EditMode) or vertex (in EditMode). Use Shift to extend the selection.
- SPACE: Display the ToolBox, a menu for creating and performing some common operations on objects.
- b: Begin border selection mode; end with left-click.
- x: Delete selection.
- g: Grab (translate) selection.
- r: Rotate selection.
- s: Scale selection.
- Shift+s: Snap (align). This operation can snap vertices or objects to the grid or cursor, or snap the cursor to objects or the grid.
- h: Hide selection (only in EditMode). This is useful for temporarily hiding parts of a large, complex object to allow interior parts to be more easily seen and edited.
- Shift+d: Create a separate duplicate copy of an object.
- Alt+d: Create a linked duplicate copy of an object. Changes to the geometry of one object are mirrored automatically in the other object.
- Ctrl+p: Create a hierarchical parent relationship between selected objects.
- Alt+p: Clear the hierarchical parent relationship between the selected objects.
- 1 through 0: Switch the currently visible layer to be a layer between 1 and 10.
- Alt+1 through Alt+0: Switch the current layer between layers 11 and 20.
- Shift+1 through Shift+0: Toggle visibility of layers 1 through 10, allowing multiple selection of layers.
- Shift+Alt+1 through Shift+Alt+0: Toggle visibility of layers 11 through 20.
- m: Move the currently selected objects to another layer.
- F1: Load file.
- F2: Save file.
- Alt+w: Export file as Videoscape, a simple ASCII format without textures.
- q: Quit Blender.

Exporting and Importing Blender Models

Models created in Blender can be imported into l3d programs. The Blender file format is very complex and not documented for external users. For this reason, we need to export the Blender models in a different file format in order to import them into l3d programs.

The simplest export format offered by Blender is the Videoscape format. This file format is simple and straightforward; it contains vertices and polygons defined in terms of indices into a common vertex list. Conveniently, this is exactly the same way that l3d stores polygonal objects.

We use a plug-in (file `vidflat.h`) to read the Videoscape file into l3d. The plug-in constructor is of particular interest. The `data` parameter passed to the constructor is a plug-in environment object of type `l3d_plugin_environment`, so that the plug-in knows something about its calling environment. The environment object, in turn, stores another `data` parameter. In the case of the Videoscape plug-in, this data parameter is a pointer to a character string containing a list of additional parameters needed to load the Videoscape file.

The Videoscape parameter string in the environment object has the following form:

```
x y z A B C D E F G H I mesh.obj
```

The entries are interpreted as follows:

- `x y z`: Floating-point numbers indicating the origin of the object's local coordinate system. In other words, this is the location of the object.
- `A B C D E F G H I`: Nine floating-point numbers representing the object's orientation as a 3 3 rotation matrix. The numbers are in row-major order (i.e., A B C is the first row of the matrix, D E F the second row, G H I the last row). Recall the relationship between matrices and coordinate systems, (as explained in the introductory companion book *Linux 3D Graphics Programming*). This relationship implies that the vector (A,D,G)—the first row of the matrix—is the *x* axis of the object's local coordinate system, (B,E,H) is the *y* axis, and (C,F,I) is the *z* axis.
- `mesh.obj`: The filename of the Videoscape file containing the geometry of the object.

The plug-in constructor first parses the parameter string to get the position, orientation, and required filename for the Videoscape object to be loaded. Next, it opens the mesh file and reads in the vertices and faces. The Videoscape format stores coordinates in a right-handed system, with the *z* axis going up; l3d uses a left-handed system with the *y* axis going up. To convert between these two systems, we need to do two things: swap the *y* and the *z* values as we read them from the Videoscape file, and reverse the ordering of the vertices in a face definition to preserve the front/back orientation of the face after the vertex swap. After converting the coordinates in this manner, the constructor then initializes each polygon by computing its center, surface normal, and containing plane. At this point, the polygon is also assigned a random color.

After reading in all vertices and faces, the constructor then defines two transformation matrices, one for rotating the object according to the rotation matrix specified within the parameter string, and a second matrix for translating the object to the specified position. These matrices are concatenated together and stored in the object as a composite transformation.



TIP See sample program `vidflat` on the CD-ROM for an example of using the Videoscape plug-in.

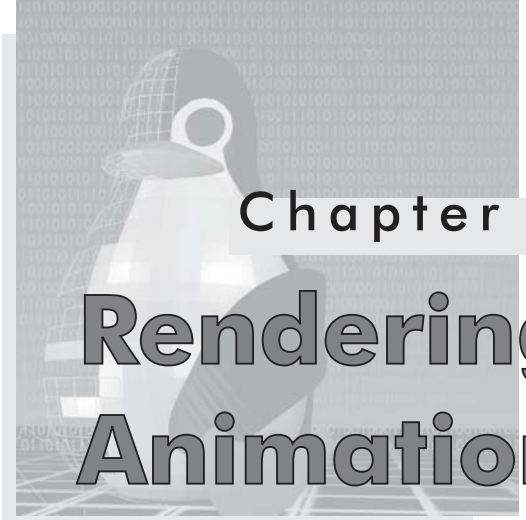
Summary

This chapter flew through the basics of Linux 3D graphics programming, as covered in depth in the introductory companion book *Linux 3D Graphics Programming*. We looked at:

- 2D graphics, theory, and practice under Linux and X
- Definition of 3D graphics
- 3D coordinate systems, vectors, and matrices
- Perspective projection
- Storing points, polygons, objects, and worlds in computer memory
- l3d library classes in C++ for implementing event-driven, object-oriented 3D graphics
- 3D modeling, export, and import with Blender

In a nutshell, that is the prerequisite knowledge for effectively reading the coming chapters. Don't worry if you are rusty on one or two concepts—the important thing to ensure, before proceeding with the rest of the book, is that you are fairly comfortable with all or almost all of the topics discussed in this chapter.

With the basics of polygonal 3D graphics now firmly in mind, we can proceed to the next chapter, which discusses rendering and animation techniques for 3D polygons.



Chapter 2

Rendering and Animation Techniques for 3D Polygons

This chapter examines rendering and animation techniques for 3D polygons. In Chapter 1, we saw that the l3d library classes provide facilities for storing, grouping, and manipulating 3D polygons, objects, and worlds. We also saw that l3d classes can rasterize flat-shaded polygons by stepping along the polygon's edges from top to bottom, and drawing horizontal spans of pixels between the left and right edges. In this chapter, we extend these basic polygonal techniques to achieve more interesting and realistic visual effects.

We cover the following topics in this chapter:

- 3D morphing (allowing a 3D object to smoothly change shape into another)
- Adding surface detail through texture mapping
- Using the symbolic algebra package Calc to derive the texture mapping equations
- Mathematical models for computing light intensities
- Rendering computed light intensities with light mapping
- Shadows

This chapter presents the following sample programs: `morph3d` illustrating vertex animation and morphing in 3D, `textest` illustrating texture mapping, and `lightmap` illustrating light mapping.

Vertex Animation and 3D Morphing

The program `morph2d` from the companion book *Linux 3D Graphics Programming* already explained and illustrated the concept of morphing in 2D. As we used the term earlier, morphing referred to the change in shape of a polygon by interpolating the vertices in its vertex list between certain key frames or morph targets.

Here, we wish to extend the concept of morphing to 3D, and also to groups of polygons instead of single polygons. As it turns out, this is extremely easy. First of all, the vertex

interpolator class we used in 2D, `l3d_vertex_interpolator`, can be reused for the 3D morphing case. Second, all polygons in a 3D object share a common vertex list; therefore, changing the vertex list in the object automatically redefines the geometry of all polygons in the object.

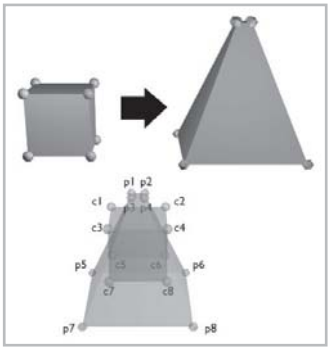


Figure 2-1: Morphing in 3D is a simple extension of the same concept in 2D. We need only interpolate the vertex list, shared by all polygons, from one key frame to another. Here, we have a cube defined by vertices c1-c8, and a pyramid defined by vertices p1-p8. Each vertex c1-c8 is morphed to the corresponding vertex p1-p8, which morphs the cube into the pyramid.

Note that in order to be completely correct, we would have to recompute the surface normal vector for all affected polygons any time we made any change to the vertex list. If we change one vertex in the vertex list, we would need to recompute the surface normals for all polygons using that vertex. If we change all vertices in the vertex list, as is usually the case with morphing, then we would need to recompute the surface normals for all polygons in the object. Also, any other geometry-specific information stored with the polygons or the object would need to be recomputed whenever we change the vertex list, because changing the vertex list is equivalent to changing the underlying geometry of the object.

In the following sample program `morph3d`, however, we won't be recomputing the normals, but rather just changing the vertex list. This leads to simpler and faster code, but is actually incorrect since the surface normals are no longer normal to the morphed geometry. Since we don't use the surface normal in this sample program, the fact that the post-morphing normals are incorrect is actually not noticeable.

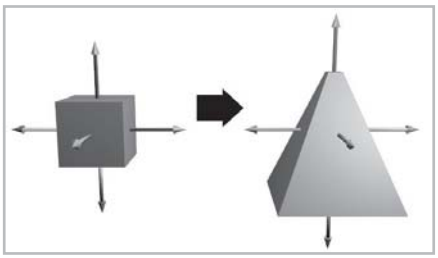


Figure 2-2: Surface normals or other additional 3D geometrical attributes are no longer accurate after morphing.

Sample Program: morph3d

Sample program `morph3d` displays a number of morphing pyramid objects in an interactive scene. Each pyramid slowly changes shape from a compact pyramid to a more elongated shape, then back again. You can interactively fly though the scene with the standard keyboard controls used in l3d library:

j:	Rotate left (rotates around the VUP axis)
l:	Rotate right (rotates around the VUP axis)
i:	Thrust forward (translates along the VFW axis)
k:	Thrust backward (translates along the VFW axis)
J:	Roll left (rotates around the VFW axis)
L:	Roll right (rotates around the VFW axis)
I:	Pitch forward (rotates around the VRI axis)
K:	Pitch backward (rotates around the VRI axis)
s:	Thrust left (translates along the VRI axis)
f:	Thrust right (translates along the VRI axis)
e:	Thrust up (translates along the VUP axis)
d:	Thrust down (translates along the VUP axis)
v:	Cycle through to the next thrust level



NOTE The VFW axis is the forward axis along which the camera is looking. The VUP axis is the axis pointing straight up from the camera. The VRI axis is the axis pointing straight to the right of the camera. Taken as a set, the vectors VRI, VUP, and VFW form a left-handed camera coordinate system.

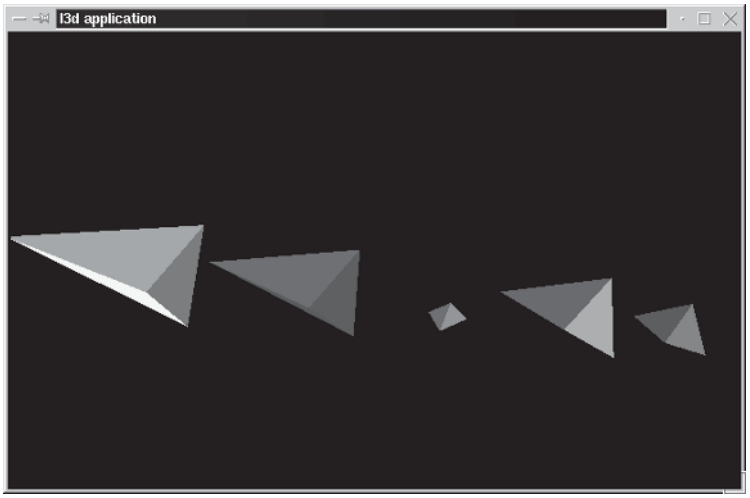


Figure 2-3: Output from program `morph3d`. Each pyramid slowly changes shape.

Listing 2-1: `main.cc`, the main file for program `morph3d`

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../lib/geom/object/object3d.h"
#include "../lib/geom/polygon/p3_flat.h"
#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rast3.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/geom/world/world.h"
```

```
#include "../lib/system/fact0_2.h"
#include "../lib/pipeline/pi_wor.h"
#include "../lib/dynamics/plugins/plugenv.h"
#include "shapes.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdarg.h>

class my_world:public l3d_world {
  public:
    my_world(void);
};

my_world::my_world(void)
    : l3d_world(640,400)
{
  l3d_screen_info *si = screen->sinfo;

  camera->VRP.set(0,0,-50,0);
  camera->near_z = float_to_l3d_real(5.5);
  camera->far_z = int_to_l3d_real(500);

  int i,j,k,onum=0;

  i=10; j=0; k=20;
  k=0;

  //- create some pyramid objects
  for(i=1; i<100; i+=20) {
    objects[onum=objects.next_index()] = new pyramid();
    l3d_screen_info_indexed *si_idx;
    l3d_screen_info_rgb *si_rgb;

    l3d_polygon_3d_flatshaded *p;
    for(int pnum=0; pnum<objects[onum]->polygons.num_items; pnum++) {
      p = dynamic_cast<l3d_polygon_3d_flatshaded *>(objects[onum]->polygons[pnum]);
      if(p) {
        p->final_color = si->ext_to_native
                         (rand()%si->ext_max_red,
                          rand()%si->ext_max_green,
                          rand()%si->ext_max_blue);
      }
    }

    if (objects[onum]==NULL) exit;
    objects[onum]->modeling_xforms[1].set
    ( int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(i),
      int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(1),
      int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(1),
      int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1) );
    objects[onum]->modeling_xform =
      objects[onum]->modeling_xforms[1] |
      objects[onum]->modeling_xforms[0] ;
  }

  screen->refresh_palette();
}
```

```
main() {
  l3d_dispatcher *d;
  l3d_pipeline_world *p;
  my_world *w;

  factory_manager_v_0_2.choose_factories();
  d = factory_manager_v_0_2.dispatcher_factory->create();

  w = new my_world();
  p  = new l3d_pipeline_world(w);
  d->pipeline = p;
  d->event_source = w->screen;

  d->start();

  delete d;
  delete p;
  delete w;
}
```

Listing 2-2: shapes.h, the header file for the 3D objects for program morph3d

```
#include "../lib/geom/object/object3d.h"
#include "geom/vertex/verint.h"

class pyramid:public l3d_object {
    static const int num_keyframes = 2;
    int keyframe_no;
    l3d_two_part_list<l3d_coordinate> *keyframes[2];
    l3d_vertex_interpolator interp;
    bool currently_interpolating;

  public:
    pyramid(void);
    virtual ~pyramid(void);
    int update(void);
};
```

Listing 2-3: shapes.cc, the implementation file for the 3D objects for program morph3d

```
#include "shapes.h"

#include <stdlib.h>
#include <string.h>

pyramid::pyramid(void) :
    l3d_object(4)
{
  keyframes[0] = new l3d_two_part_list<l3d_coordinate>(4);
  (*keyframes[0])[0].original.set
  (float_to_l3d_real(0.),
   float_to_l3d_real(0.),
   float_to_l3d_real(0.),
   float_to_l3d_real(1.));
  (*keyframes[0])[1].original.set
  (float_to_l3d_real(10.0),
   float_to_l3d_real(0.),
   float_to_l3d_real(0.),
   float_to_l3d_real(1.));
  (*keyframes[0])[2].original.set
  (float_to_l3d_real(0.),
   float_to_l3d_real(10.),
```

```
 float_to_l3d_real(0.),
 float_to_l3d_real(1.));
(*keyframes[0])[3].original.set
(float_to_l3d_real(0.),
 float_to_l3d_real(0.),
 float_to_l3d_real(10.),
 float_to_l3d_real(1.));

keyframes[1] = new l3d_two_part_list<l3d_coordinate>(4);
(*keyframes[1])[0].original.set
(float_to_l3d_real(5.),
 float_to_l3d_real(5.),
 float_to_l3d_real(5.),
 float_to_l3d_real(1.));
(*keyframes[1])[1].original.set
(float_to_l3d_real(-10.),
 float_to_l3d_real(10.),
 float_to_l3d_real(0.),
 float_to_l3d_real(1.));
(*keyframes[1])[2].original.set
(float_to_l3d_real(10.),
 float_to_l3d_real(0.),
 float_to_l3d_real(0.),
 float_to_l3d_real(1.));
(*keyframes[1])[3].original.set
(float_to_l3d_real(10.0),
 float_to_l3d_real(15.0),
 float_to_l3d_real(0.),
 float_to_l3d_real(1.));

int pi;
pi = polygons.next_index();
polygons[pi] = new l3d_polygon_3d_flatshaded(3);
printf("before: %p", polygons[pi]->vlist);
polygons[pi]->vlist = &vertices;
printf("after: %p", polygons[pi]->vlist);
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=0;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=1;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=3;
polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();

pi = polygons.next_index();
polygons[pi] = new l3d_polygon_3d_flatshaded(3);
polygons[pi]->vlist = &vertices;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=2;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=3;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=1;
polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();

pi = polygons.next_index();
polygons[pi] = new l3d_polygon_3d_flatshaded(3);
polygons[pi]->vlist = &vertices;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=0;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=2;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=1;
polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();

pi = polygons.next_index();
polygons[pi] = new l3d_polygon_3d_flatshaded(3);
polygons[pi]->vlist = &vertices;
```

TEAMFLY

Team-Fly®

```
  (*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=3;
  (*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=2;
  (*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=0;
  polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();

  num_xforms = 2;
  modeling_xforms[0] = l3d_mat_rotx(0);
  modeling_xforms[1].set
  ( float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.),
    float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.),
    float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.),
    float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.) );

  modeling_xform=
    modeling_xforms[1] | modeling_xforms[0];

  currently_interpolating = false;
  keyframe_no=0;
  vertices = keyframes[keyframe_no];
}

pyramid::~pyramid(void) {
  for(register int i=0; i<polygons.num_items; i++) {delete polygons[i]; }
  for(int i=0; i<num_keyframes; i++) {
    delete keyframes[i];
  }
}

int pyramid::update(void) {
  if(currently_interpolating) {
    vertices = interp.list;
    if(! interp.step()) {
      currently_interpolating = false;
    }
  }else {
    keyframe_no++;
    if(keyframe_no >= num_keyframes) {keyframe_no = 0; }
    int next_keyframe = keyframe_no + 1;
    if(next_keyframe >= num_keyframes) {next_keyframe = 0; }

    vertices = keyframes[keyframe_no];
    interp.start( *keyframes[keyframe_no], *keyframes[next_keyframe],
                  rand()%100 + 50, 3);

    currently_interpolating = true;
  }
}
```

The program `morph3d` uses the `l3d_world` and `l3d_object` classes to store the virtual world. The file `main.cc` is very simple: it declares a world subclass which creates some `pyramid` objects.

The `pyramid` class is a morphing 3D pyramid. It changes its shape between a compact and an elongated pyramid. To accomplish this, we added the following lines to the `pyramid` class:

```
class pyramid:public l3d_object {
    static const int num_keyframes = 2;
    int keyframe_no;
    l3d_two_part_list<l3d_coordinate> *keyframes[2];
    l3d_vertex_interpolator interp;
    bool currently_interpolating;
```

The `keyframes` variable holds two separate, different vertex lists. Each list is one complete set of vertex positions defining a shape for the 3D object. We create and fill these lists within the `pyramid` constructor.

```
keyframes[0] = new l3d_two_part_list<l3d_coordinate>(4);
(*keyframes[0])[0].original.set
(float_to_l3d_real(0.),
 float_to_l3d_real(0.),
 float_to_l3d_real(0.),
 float_to_l3d_real(1.));
(*keyframes[0])[1].original.set
(float_to_l3d_real(10.0),
 float_to_l3d_real(0.),
 float_to_l3d_real(0.),
 float_to_l3d_real(1.));
(*keyframes[0])[2].original.set
(float_to_l3d_real(0.),
 float_to_l3d_real(10.),
 float_to_l3d_real(0.),
 float_to_l3d_real(1.));
(*keyframes[0])[3].original.set
(float_to_l3d_real(0.),
 float_to_l3d_real(0.),
 float_to_l3d_real(10.),
 float_to_l3d_real(1.));

keyframes[1] = new l3d_two_part_list<l3d_coordinate>(4);
(*keyframes[1])[0].original.set
(float_to_l3d_real(5.),
 float_to_l3d_real(5.),
 float_to_l3d_real(5.),
 float_to_l3d_real(1.));
(*keyframes[1])[1].original.set
(float_to_l3d_real(-10.),
 float_to_l3d_real(10.),
 float_to_l3d_real(0.),
 float_to_l3d_real(1.));
(*keyframes[1])[2].original.set
(float_to_l3d_real(10.),
 float_to_l3d_real(0.),
 float_to_l3d_real(0.),
 float_to_l3d_real(1.));
(*keyframes[1])[3].original.set
(float_to_l3d_real(10.0),
 float_to_l3d_real(15.0),
 float_to_l3d_real(0.),
 float_to_l3d_real(1.));
```

Now, given two different shapes of the object stored in two separate vertex lists, we use the class `l3d_vertex_interpolator`, exactly as we did in the earlier program `morph2d`, to interpolate the positions of the vertices between the two vertex lists. This takes place in the method `pyramid::update`. The usage of the vertex interpolator is exactly the same as before: we specify a starting vertex list, an ending vertex list, a total count of interpolation steps, and a dimension (2D or 3D) determining which elements of the vertex are interpolated. The vertex interpolator has its own internal, separate vertex list, which contains the interpolated "in-between" positions between the specified starting vertex list and ending vertex list. We interpolate between vertex lists by calling `interp.step`, and assign the interpolated vertex list to the pyramid's vertex list.

```
if(currently_interpolating) {
  vertices = interp.list;
  if(! interp.step()) {
    currently_interpolating = false;
  }
```

Once the interpolation has finished, after the specified number of steps have been taken and the positions of the vertices in the vertex list have reached the positions in the target vertex list, we then proceed to interpolate between the next two pairs of vertex lists in the array `keyframes`. In this program, since we only have two vertex lists, this simply has the effect of morphing back from the second to the first vertex list again. Notice that we pass to the vertex interpolator a random parameter for the number of interpolation steps, so that each pyramid always morphs at a different speed.

```
}else {
  keyframe_no++;
  if(keyframe_no >= num_keyframes) {keyframe_no = 0; }
  int next_keyframe = keyframe_no + 1;
  if(next_keyframe >= num_keyframes) {next_keyframe = 0; }

  vertices = keyframes[keyframe_no];
  interp.start( *keyframes[keyframe_no], *keyframes[next_keyframe],
                rand()%100 + 50, 3);

  currently_interpolating = true;
}
```

That's all there is to morphing in 3D. It's nothing more than a step-by-step interpolation of all vertex positions from a starting list to an ending list, just as with the 2D case.



NOTE When morphing more complicated shapes, you will want to use a 3D modeling program (also called a 3D modeler) to create the 3D objects. Chapter 3 covers using Blender to do exactly this. To create models which morph into one another, it is vital that both models have the same number of vertices and faces in the same order. The best way to ensure this is simply to start with the first model and modify it within the 3D modeler to become the second, target model. If you create the target model from scratch, it is almost certain that the number or order of the vertices and faces will be different from those of the first model, which would make morphing impossible or visually disturbing. See Chapter 3 for more details.



NOTE It is actually not "impossible" to morph between two objects with different numbers of vertices or faces, but doing so is much more complicated than the case described above (where a 1:1 correspondence does exist between the vertices of the two objects). You would have to define some function to automatically map vertices to "reasonable" in-between positions, based on the shapes of the source and target objects, a topic which goes beyond the scope of this book.

Lighting

Polygons, as we have treated them until now, have all been single-colored. We'll now look at some methods of combining color and light so that our polygons look more realistic.

For our purposes, we model light as a single, integer intensity value which changes the brightness of a particular color. Therefore, we combine light and color to arrive at a final lighted color. This is accomplished by means of lighting tables. In the lighting table, we simply look up the color along one axis, and the light intensity along the other axis, and find an entry in the table that tells us the final color with the specified intensity. For colors specified directly with red, green, and blue values (as opposed to through a palette), we split the lighting tables into three separate red, green, and blue tables, solely for the reason of saving memory.

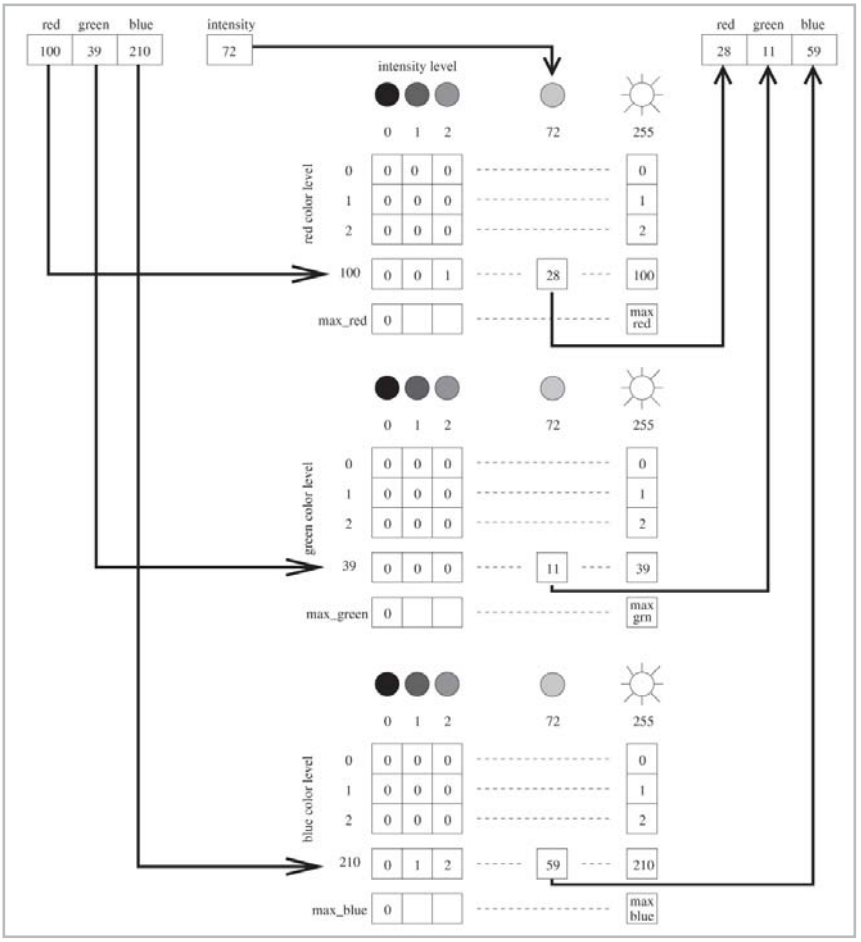


Figure 2-4: Red, green, and blue lighting tables in use.

We define intensities as going from 0 to a certain predefined maximum integer light level, `MAX_LIGHT_LEVELS`. A light intensity of 0 combined with any color yields black; a light intensity of `MAX_LIGHT_LEVELS` combined with a color yields the original color at full intensity.

The light model just described assumes that the light itself has a color of white. It is also possible to model colored light sources, which not only change the intensity of the color in question, but also change the color itself by introducing new color components contained within the light itself. For instance, shining a red light on a white surface yields a final color of red. This can be implemented most easily using an RGB color model. The color of the light is represented as a red intensity, a green intensity, and a blue intensity. The color of the surface is represented as a red color, a green color, and a blue color. The color of the final, lighted surface is the red color lighted by the red intensity, combined with the green color lighted by the green intensity, combined with the blue color lighted by the blue intensity.

We can view the lighting problem as two separate issues. The first issue is computing the light intensity itself. The second issue is deciding where to compute and draw the light intensities (once per polygon, once per vertex, once per pixel, and so forth). We now cover each of these topics separately, starting with the computation of light intensities.

Mathematical Models for Computing Light

We are physically able to see objects because they emit or reflect light energy. There are various mathematical models we can use to model the amount of light emitted by or reflected from a surface.

The lighting models presented in the next few sections attempt to approximate the effects of light by computing the amount of light we see from a surface. Note that these lighting models do not necessarily represent a realistic modeling of the physical phenomena underlying the emission and reflection of light, but nevertheless provide adequate results in practice and are widely used in interactive 3D graphics.

The lighting equations presented below all assume that light intensities are real values which go from 0 to 1. We can then multiply these with a scaling factor to change them into integer values to use as lookup indices into the light table. In the l3d library, the scaling factor is called `MAX_LIGHT_LEVELS` and has a value of 255.

Self Lighting

One simple lighting model is to assume that each object emits its own light energy and reflects no light energy. In this case, each object's light intensity is determined solely by its own inherent brightness; it does not depend on any external factors. We can express the equation for self lighting as follows:

Equation 2-1 $$I_{self} = I_o$$

In this equation, I_{self} represents the computed intensity based on the self lighting model, and I_{o} represents the inherent brightness of the object itself.

While self intensity is an easy lighting model to represent, it is more interesting to model the reflection of light off of surfaces, since reflected light is the majority of light we see in a typical real-world scene.

Ambient Lighting

The term *ambient light* refers to light which has been scattered and reflected so many times, off of so many objects, that its original direction cannot be determined. Ambient light, therefore, is nondirectional light which equally reflects off of and equally illuminates all objects. It also provides a convenient way within a scene of modeling a global minimum light level whose exact source is unimportant. We can express ambient lighting as a simple global constant for the entire scene:

Equation 2-2 $$I_{amb} = I_a$$

We could also multiply the ambient light intensity by a scaling factor, ranging from 0 to 1, representing the reflectiveness of the surface to which the ambient light is applied. This can have the effect of reducing the effect of the ambient light if the surface is less reflective. The ambient scaling factor would be stored with each object or with each polygon.

Equation 2-3 $$I_{amb} = I_a M_a$$

Such parameters particular to a surface which affect the incident light are called *material parameters*.

Diffuse Reflection

Ambient light is still not a very satisfying light model because it affects all surfaces identically. We may obtain a better result by modeling the diffuse reflection of light off of a surface. Matte surfaces, also called diffuse surfaces, scatter light rays with equal probability in all directions, resulting in a diffuse illumination of the surface.

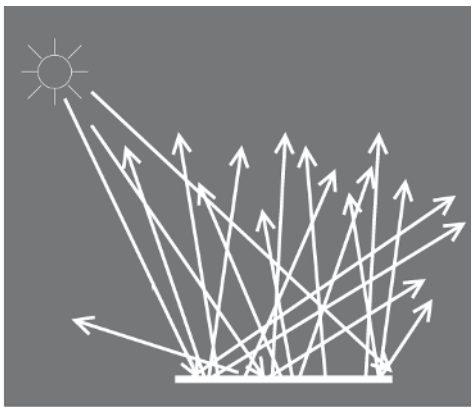


Figure 2-5: Diffuse reflection of light rays.

Mathematically, let us assume that we have a *point light source* and a surface for which we wish to compute the diffuse illumination. A point light source is an infinitely small emitter of light energy located at a single point some finite distance from the surface, which casts rays of light energy (consisting of photons) equally in all directions. To compute the diffuse illumination of the surface from the point light source, we must compute two quantities. The first is the amount of

light striking the surface. The second is the amount of light that the viewer sees reflected off of the surface.

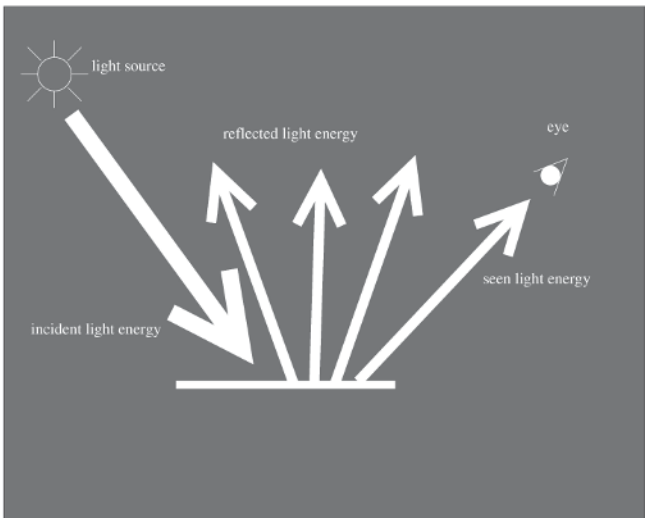


Figure 2-6: A point light source, a surface, and the viewer. We are interested in two quantities: how much the light illuminates the surface, and how much of the light the viewer sees reflected off of the surface.

We compute the amount of light striking the surface by considering an infinitesimally small beam of light energy striking a surface. For a constant light intensity, this light beam imparts a certain fixed amount of light energy to the surface it strikes. It turns out that the angle that the light beam forms with the surface normal vector is important. If the light beam strikes the surface directly head-on, then the resulting light intensity on the surface is at its maximum. If the light beam strikes the surface at an angle, the light intensity on the surface decreases. This is because the amount of surface area covered by the light beam is greater when it strikes the surface at an angle, and thus the fixed quantity of light energy from the light beam is divided across a larger surface area, yielding an overall lesser intensity per unit area of surface.

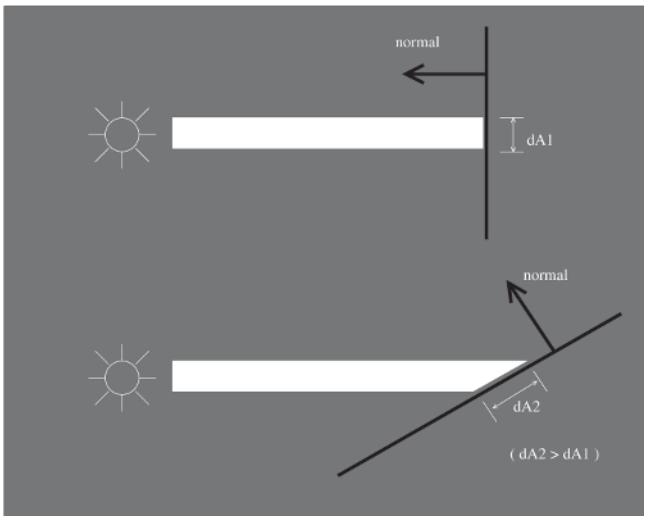


Figure 2-7: A light beam covers the least surface area when striking the surface directly along its normal vector; as the angle to the normal vector increases, the beam must cover a larger surface area, resulting in decreased overall intensity.

Mathematically, we can express this relationship as follows.

Equation 2-4 $$I_{diff} = I_p cos\theta$$

I_p is the intensity of the point light source, ranging from the darkest value of 0 to the brightest value of 1. The angle theta is the angle between the incident light ray and the surface normal. If the angle is zero, then the light strikes the surface head-on, the cosine of the angle is 1, and the intensity is maximum. As the angle approaches 90 or –90 degrees, the cosine approaches zero, causing the light intensity to decrease.

We can also formulate the diffuse lighting equation as follows.

Equation 2-5 $$I_{diff} = I_p L \cdot N$$

Here, N is the surface normal vector, and L is the vector from the surface to the light source. Note that both vectors must be normalized. Recall that the dot product of these two normalized vectors is exactly the same as computing the cosine of the angle between them.

We can also multiply the computed light by a diffuse scaling factor M_d, ranging from 0 to 1 depending on the diffuse reflectiveness of the material, to reduce the effect of the light for less reflective materials.

Equation 2-6 $$I_{diff} = I_p M_d L \cdot N$$

Finally, we can also multiply the light intensity by an attenuation factor f_{att} between 0 and 1, so that the computed light decreases with its distance from the light source, in accordance with the physical reality that the amount of light energy reaching a surface decreases with distance.

Equation 2-7 $$I_{diff} = f_{att} I_p M_d L \cdot N$$

It is suggested in *Computer Graphics: Principles and Practice* that this distance-based falloff factor be computed as follows [FOLE92].

Equation 2-8 $$f_{att} = min(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}), 1$$

Term d_L represents the distance between the light source and the surface being illuminated, and the terms c_1, c_2, and c_3 are user-defined constants for the light source. The attenuation factor is therefore inversely proportional to the distance, but is never greater than one (otherwise it would increase, not decrease, the effect of the light). The constant c_1 is chosen to prevent the denominator from being too small when d_L is very small, and the constants c_2 and c_3 allow us to control whether the light falls off proportional to the inverse square of the distance, or to the inverse distance, or to a combination of both. Physically, for a single point light source, the energy falls off as the inverse square of the distance, but typical objects in real-world scenes receive light from more than just a single point light source. This means that the inverse square energy falloff is much too sudden, which is the reason that we introduce the c_1 and $c_2 d_L$ terms in the denominator, allowing for a more gradual light falloff.

Now, given the computed light intensity that reaches a surface from a point light source, the second question we would like to answer is how much of this light a viewer sees from a particular

viewing position relative to the surface. This depends on how much light is reflected from the surface.

As it turns out, the viewing position ends up having no effect on the final light intensity seen off of a diffuse surface. This means that the diffuse light intensity as computed above represents the amount of light seen reflected off of the surface, from any viewing angle or position. Let us examine why this is so.



TIP Diffuse illumination depends only on the positions and orientations of the light source and the surface. It does not depend on the position of the camera.

As we mentioned earlier, the amount of light the viewer sees from a surface depends on the amount of light reflected by the surface. Matte surfaces, also referred to as *Lambertian surfaces*, obey a law of physics called Lambert's law. Lambert's law states that, ideally, diffuse surfaces reflect light, per unit area, directly proportional to the cosine between the surface normal vector and the vector going from the surface to the viewer (call this last vector the view vector). This means that if we consider a particular fixed amount of the surface, the greater the angle between the normal vector and the view vector, the less reflected light we see. This would at first seem to imply that the light intensity should decrease as the angle increases. But this is not the case, because as the angle between the normal vector and the view vector increases, we also see more of the surface—in particular, we see proportionally more of the surface according to the reciprocal of the cosine of the angle between the normal and the view vector. The increase in seen surface area exactly cancels out the decrease in light intensity per unit area. In other words, as the angle increases, the reflected light energy per unit area decreases, but the amount of area we see increases by exactly the same proportion, meaning that the amount of reflected light stays the same regardless of the viewer's position, for Lambertian surfaces.

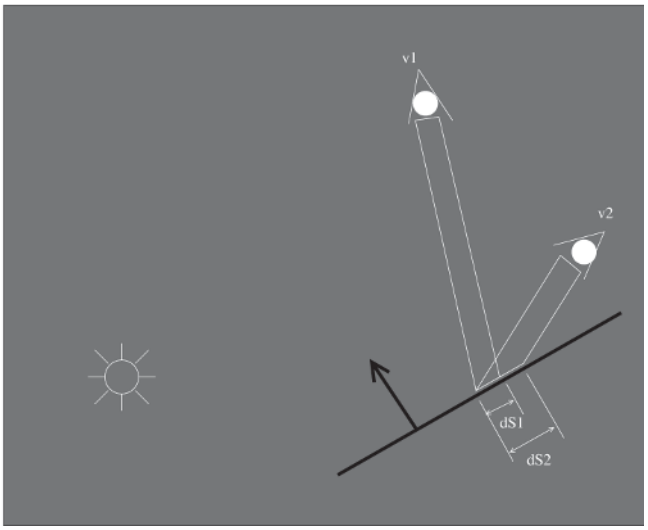


Figure 2-8: As the angle between the viewer and the diffuse surface increases (from viewpoint v1 to v2), the reflected light decreases according to the cosine of the angle (Lambert's law). But geometrically, we simultaneously see more of the surface, according to the reciprocal of the cosine of the angle (dS2 > dS1). The two factors cancel, yielding a viewpoint-independent light intensity.

A few comments about the fundamental assumptions of the preceding light model are in order. The light model we just described works by computing the diffuse illumination from all light sources off of all surfaces. This implies a strict separation between light sources and the surfaces being illuminated. This model is not complete because it does not account for the inter-object reflections that take place in the real world. An object reflecting light then also becomes a light source, further illuminating other objects, which again reflect light. One way to simulate the effect of this complex inter-object illumination is simply to add an ambient term to the overall lighting; since the inter-object light reflection is so complex, the resulting illumination can be considered as without direction and ambient. A more accurate, though much more computationally intensive, means of accounting for inter-object illumination is radiosity, which is covered below.

Another assumption we made was that our light sources were point light sources, which are located at a particular point in space and radiate light energy equally in all directions. Alternative models of light sources are also possible. Sunlight illuminating objects on Earth, for instance, is not really accurately modeled by a point light source, because the distance of the sun from the earth is practically infinite in comparison to the dimensions of the objects being illuminated. Because of the great distance separating the sun and the earth, all the light rays coming from the sun are practically parallel to one another. This means that in the lighting equations presented above, the vector L from the surface to the light source always points in the same direction for all polygons, in the direction of the parallel rays of sunlight.

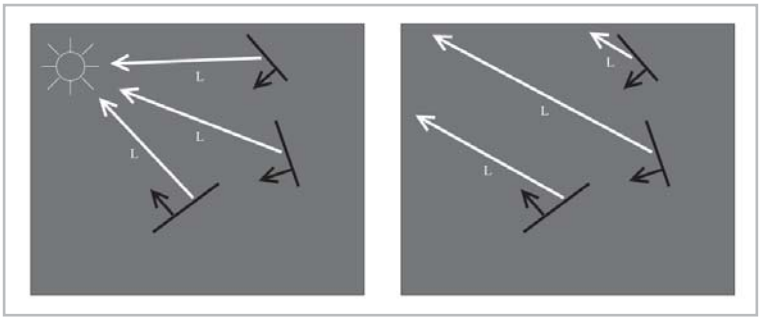


Figure 2-9: For a point light source, the vector L points from the surface to the light source; each different surface orientation yields a different vector L (left). For sunlight, which can be thought of as an infinitely distant point light source, the vector L is the same for all surface orientations (right).

Finally, it is also possible to model spotlights, which have only an exactly defined area of influence, such as a cone. Outside of the area of influence the light does not affect the surface's intensity at all. This allows for creating illumination effects on the surface with sharply defined edges, much as a real-world spotlight only illuminates within a certain elliptical area on the ground.



Figure 2-10: Spotlights clearly delineate the area of influence of the light. The left surface is illuminated with a point light; the right, with a spotlight.

Team-Fly®

Specular Reflection

Specular reflection refers to the bright highlights which appear when a shiny surface is illuminated. Such effects are not accounted for by the preceding lighting equations.

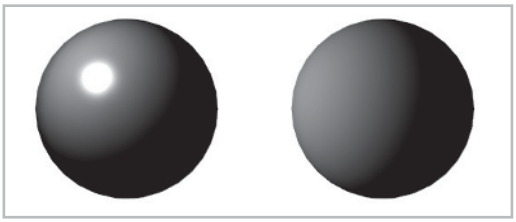


Figure 2-11: The left sphere has been rendered with a specular highlight; the right, only with diffuse lighting.

There are a number of ways of computing the specular reflection. One which we examine here is the so-called Phong illumination model. Do not confuse Phong *illumination*, which deals with the computation of specular light intensities, with Phong *shading*, which refers to a rendering method (covered in the next section) for drawing already computed light intensities.

Phong illumination assumes the surface is shiny and reflective. Therefore, with Phong illumination, we compute a *reflection vector*, which is the reflection of the light vector *L* (going from the surface to the light source) around the surface normal vector *N*.

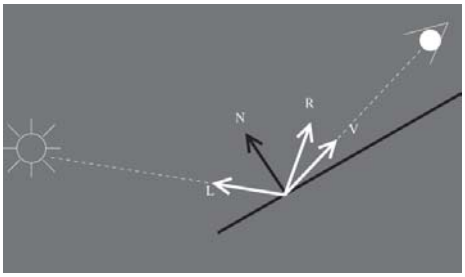


Figure 2-12: The reflection vector R for Phong illumination.

The idea is that as the orientation of this reflection vector *R* comes nearer to the orientation of the view vector *V* (going from the surface to the viewer), the amount of reflected light should increase exponentially, causing a sudden, bright highlight to appear.

Mathematically, we can express Phong illumination as follows. Note that all vector quantities are normalized.

Equation 2-9 $$I_{spec} = (cos\alpha)^n = (R \cdot V)^n$$

Thus, the specular intensity increases exponentially as the angle alpha between the reflection vector and the view vector decreases. The exponential cosine term has this effect; if the vectors lie on top of one another, the cosine is one, and the intensity is at a maximum. As the cosine decreases, the intensity falls off rapidly. The cosine of the angle can be computed by dotting the reflection vector with the view vector. The exponential factor *n* is a parameter describing the shininess of the surface; the greater the value of *n*, the shinier the surface appears because of an exponentially faster increase and decrease of light intensity around the highlight.

The only issue which remains is how to find the reflection vector. We can calculate the reflection vector by the following equation.

Equation 2-10
$$R = (2(N \cdot L))\, N - L$$

The easiest way to derive this is to use the rotation matrix for rotation about an arbitrary axis. Here, we wish to rotate the vector *L* by 180 degrees about vector *N*. Recall the rotation matrix for rotation about an arbitrary axis:

Equation 2-11
$$\begin{bmatrix} u_1^2 + cos\theta(1-u_1^2) & u_1u_2(1-cos\theta) - u_3 sin\theta & u_3u_1(1-cos\theta) + u_2 sin\theta & 0 \\ u_1u_2(1-cos\theta) + u_3 sin\theta & u_2^2 + cos\theta(1-u_2^2) & u_2u_3(1-cos\theta) - u_1 sin\theta & 0 \\ u_3u_1(1-cos\theta) - u_2 sin\theta & u_2u_3(1-cos\theta) + u_1 sin\theta & u_3^2 + cos\theta(1-u_3^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The elements of the matrix are the components of the vector about which the rotation is to take place, in this case *N*. Replacing all *u* entries with corresponding *N* entries, multiplying the matrix with the vector *L* to be rotated, and realizing that the cosine of 180 degrees is –1 and the sine of 180 degrees is 0, gives the following expression:

Equation 2-12
$$\begin{bmatrix} N_1^2 - (1-N_1^2) & N_1N_2(2) & N_3N_1(2) & 0 \\ N_1N_2(2) & N_2^2 - (1-N_2^2) & N_2N_3(2) & 0 \\ N_3N_1(2) & N_2N_3(2) & N_3^2 - (1-N_3^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ 0 \end{bmatrix}$$

Simplifying further yields:

Equation 2-13
$$\begin{bmatrix} 2N_1^2 - 1 & 2N_1N_2 & 2N_3N_1 & 0 \\ 2N_1N_2 & 2N_2^2 - 1 & 2N_2N_3 & 0 \\ 2N_3N_1 & 2N_2N_3 & 2N_3^2 - 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ 0 \end{bmatrix}$$

Performing the matrix multiplication gives us:

Equation 2-14
$$\begin{bmatrix} 2N_1N_1L_1 - L_1 + 2N_1N_2L_2 + 2N_3N_1L_3 \\ 2N_1N_2L_1 + 2N_2N_2L_2 - L_2 + 2N_2N_3L_3 \\ 2N_3N_1L_1 + 2N_2N_3L_2 + 2N_3N_3L_3 - L_3 \\ 0 \end{bmatrix}$$

Notice that each row contains a subtraction with an isolated *L* term; thus, we can we can pull out the entire vector $[L_1,L_2,L_3,0]^T$ and express this as a vector subtraction. We can also isolate the vector $[N_1,N_2,N_3,0]^T$. Rearranging the equation to be the difference of these two vectors yields the following expression:

Equation 2-15
$$2(N_1L_1 + N_2L_2 + N_3L_3) \begin{bmatrix} N_1 \\ N_2 \\ N_3 \\ 0 \end{bmatrix} - \begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ 0 \end{bmatrix}$$

which simplifies to:

Equation 2-16 $$(2(N \cdot L))\,N - L$$

For a geometrical derivation of the reflection vector, see *Computer Graphics: Principles and Practice* [FOLE92].

Note that unlike diffuse reflection, the effect of which was independent of the viewer position, specular lighting depends on the viewer position, because the position of the highlight changes as the viewer looks at the object from a different angle or position.



NOTE Specular lighting, when it is unwanted, can cause some odd lighting artifacts. I encountered this once while programming a virtual reality application for the CAVE environment (an immersive 3D environment which surrounds the viewer on all sides with four huge screens and provides for true stereoscopic perception and head tracking through special shutter glasses). One odd problem that cropped up was with the floor of a room we had modeled. The surface of the floor appeared to change intensity suddenly when the viewer was located at particular positions within the world. Changing the position, number, and type of light sources did not eliminate the problem. It turned out that the problem was the 3D model exported by the 3D artist included a specular lighting component, which then caused the undesired camera-dependent specular "flashes" of light on the floor. These were particularly problematic because the floor, which was not intended to be specular, was not highly tessellated, meaning that the specular light computation was applied to very large triangles. This caused very large and intermittent white flashes. The solution was to remove the specification of the specular component in the exported 3D model.

Multiple Light Sources and Components

If we wish to compute the contributions of several light sources to a surface's illumination, we merely need to compute the light contribution of each light source, then add all the contributions together. Similarly, if we wish to compute lighting based on several light components (for instance, a combination of self-emitted light energy, diffuse reflected light, specular reflection, and ambient light) we also merely need to compute each light contribution separately, then add them all together to arrive at the final light intensity.

One practical problem which may arise is that after summing the light contributions from various sources, the resulting light value may be greater than the maximum allowable intensity of 1.0, which, when multiplied with the integer scaling factor mentioned at the beginning of the discussion on lighting, would yield a lighting table index that would be greater than the maximum allowable index of `MAX_LIGHT_LEVELS`. In such a case, we can clamp the light intensity to the maximum value, at the cost of some inaccuracy (the intensity should actually be brighter than the maximum possible value). You can see an example of such clamping in Figure 2-11; there, the specular highlight should not be a circle of equal white intensity, instead getting brighter towards its center, but it is impossible for the color on the printed page to increase brighter than pure white. The human eye also has a physically limited maximum input level, so clipping of input light levels also occurs naturally.

Radiosity and Ray Tracing

Radiosity is a very realistic technique for computing light intensities, based on a physical model of light energy. It works by modeling the physical interaction of light with surfaces, and by ensuring that all of the light energy in a scene is accounted for. Key to the radiosity method is the fact that it also accounts for the illumination resulting from the reflection of light off of surfaces; the previous methods all consider light sources separately from the illuminated surfaces. Radiosity generally works by dividing an entire environment into a number of small patches, each of which is characterized by the amount of light energy it emits or reflects. This implies that all parts of the model can potentially emit light, providing for correct inter-object lighting. The difficult part about radiosity is that the entire lighting solution across all patches must be consistent, meaning that for a large number of patches, a large number of lighting equations must be simultaneously solved to arrive at the correct intensity for each patch. This only makes sense; if each patch can emit light and affect all other patches, which then also emit different amounts of light, a consistent solution requires satisfying the lighting equation for every patch simultaneously. A popular method of computing radiosity is a *progressive refinement* method, where the original model is subdivided into patches only as necessary; if a large change in light intensity is observed for a particular patch, then the patch is subdivided, and the light is recomputed for each smaller patch.

As realistic as radiosity is, it is slow to compute. Therefore, its role in interactive 3D graphics is more of a preprocessing step. Radiosity can be used in an offline manner to compute light levels, which are then saved along with the 3D geometry. Then, at run time, we simply draw the light using the precomputed light levels.

Ray tracing is a rendering technique that can be used to model both reflection and refraction of light. The technique works by tracing the paths of "reverse" light rays from the eye, through each pixel in the image, and to an object in the scene (ordinarily light travels in the other direction, from the objects to the eye). The reverse light rays, like real light rays, are allowed to bounce among objects, and to be refracted after traveling through objects. This means that ray tracing can also create very realistic lighting effects. Ray tracing is currently too slow for real-time applications. See Chapter 7 for more information on ray tracing, and a way of simulating some of the effects of ray tracing through environment mapping.

Dynamic or Static Lighting Computations

One question we have not addressed is exactly when we carry out the lighting calculations. This answer is partially dependent on the rendering technique we use for drawing the polygons (covered in the next section); it is also partially dependent on whether we want *static lighting* or *dynamic lighting*. Static lighting is lighting which does not change throughout the course of a user's interaction with the virtual environment. In such a case, we compute light intensities in a preprocessing stage and store these light intensities in some format needed by our rendering technique; storing one color per polygon, or one color per vertex, or one color per lumel in a light map are all possible means of saving the computed light intensities. Then, when drawing the interactive 3D scene, we do not need to worry about computing any light intensities; the light has already been "burned into" or "painted onto" our geometry, and only needs to be drawn using an appropriate rendering technique.

Dynamic lighting is lighting that changes while a user is interacting with the 3D environment. Dynamic lighting implies a change in any factor affecting any of the lighting equations used. Such factors include surface geometry, surface orientation, surface attributes, light position, light attributes, and lighting parameters. Based on the new parameters, we dynamically recompute the illumination for all affected parts of all geometry, and store the new illumination values with the geometry for later use by the light rendering process. By defining lights to only have a certain area of influence (such as a sphere), we can limit the amount of geometry that dynamic light affects and therefore limit the amount of computation that needs to be done when this dynamic light changes intensity.

Fog

Fog is not really a lighting effect, but it affects the intensity of the surface being viewed, so we consider it here as well. Fog simply blends an original color with a certain fog color, where the amount of the blending is based on the distance of the surface to the viewer. Objects very far away from the viewer fade into the fog and are thus displayed with the fog color itself. Near the viewer, the effect of fog is negligible, and thus the surface should be displayed with its original color. From near to far the colors should fade gradually into the fog.

Mathematically, we can express the effect of fog on a particular surface color as follows.

Equation 2-17 $$c = (1 - f)c_s + f\,c_f$$

Here, c represents the final color of the surface with the effect of fog, c_S is the original, unfogged color of the surface, c_f is the color of the fog, and f is the *fog factor*, which ranges from 0 to 1. If f is 0, no fog is applied and we see the original color. If f is 1, we see only the fogged color. Between 0 and 1 are varying amounts of fog. The way we actually implement this in the C++ code in the l3d library is to scale the real-valued fog factor to lie within the integer range from zero to `MAX_LIGHT_LEVELS`. We then use the integer scaled fog factor as a lookup index into a precomputed fog table which slowly fades each color or color component (red, green, or blue) from its original value into the fog color of white.



CAUTION The OpenGL definition of the fog factor reverses the sense of 0 and 1: a fog factor of 0 in OpenGL means that we see only the fog color; a fog factor of 1 means that we see only the original color.

The question remains as to how to compute the fog factor f. The most correct way would be to use *radial fog*, which computes the distance of each fogged surface to the viewer and uses this actual distance as the basis of some blending function going from 0 to 1. But distance computations require a time-consuming square root operation, and since fog is typically computed per pixel, this can be slow. A simpler fog computation, which is less accurate, is based on the z values of each pixel after the perspective transformation. The z values are then mapped to the range 0 to 1. See Figure 2-13 for a comparison of radial fog versus z depth fog.

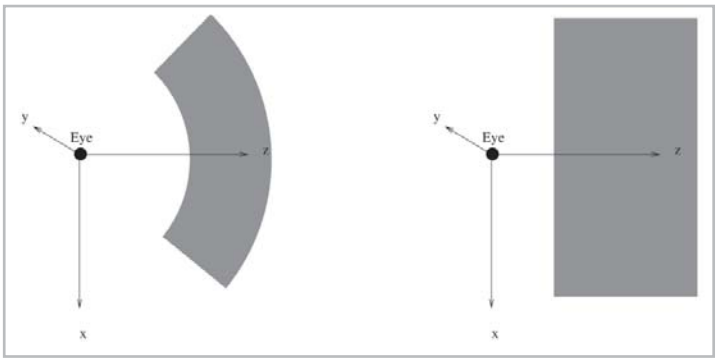


Figure 2-13: Correct radial fog versus simpler z depth fog.

Having decided on a distance to use (radial distance or *z* depth distance), we then need to map this distance into a fog value from 0 to 1. We can choose to map the fog linearly or exponentially. A linear mapping causes colors to fade out at a constant rate as they move into the fog. An exponential mapping causes colors to fade out more quickly as they recede into the fog. We can even use a discontinuous fog function to cause fog only to appear at certain distances from the viewer.



NOTE An option somewhere between z depth fog and radial fog is to approximate radial fog by taking the z depth value, then subtracting a percentage of *x* and *y* deviation from the center of the screen. This is more accurate than z depth fog, but still faster than true radial fog.

Notice that regardless of the means of computing fog, the computation is done in camera coordinates because fog is an effect which depends on distance from the camera. This is another example of why we need the `camera_transform` method in classes `l3d_object` and `l3d_polygon_3d`, which allows us to save the camera space coordinates for later use. The fog computation is typically done just before polygons are about to be drawn to the screen in the rasterizer; at this point, however, the polygons have already undergone the perspective transformation, meaning that we must have previously saved the camera space coordinates in order to perform this late fog computation.

Rendering Techniques for Drawing Light

The previous sections dealt with the mathematical computation of light intensities based on the properties and orientations of the light sources and the surfaces. We haven't yet addressed the issue of how often and exactly at which points in 3D space we need to evaluate the lighting equations. It turns out that this issue depends on the rendering technique we use for drawing light on our polygons. Such rendering techniques are also called *shading* techniques.

Let's now look at a number of rendering techniques for light, and also at the implications that each technique has on the frequency and location of lighting computations.



TIP The rendering or shading technique influences how many lighting calculations we perform and at which points in 3D space.

Flat Shading

Flat shading is the rendering technique we have been using so far: each polygon has one color.

With flat shading, we perform the lighting computations once per polygon, typically for the center point of the polygon. This computed light intensity is then used for the entire polygon.

Flat shading leads to a faceted appearance of the model, because the light intensity changes abruptly from one face to another. See Figure 2-14.



Figure 2-14: Flat shading.

Gouraud Shading

Gouraud shading is a simple technique for combating the faceted appearance of flat-shaded polygons. With Gouraud shading, we perform the lighting computations once per vertex for each polygon. This means that we need to store and transform vertex normals along with the geometry. Given the light values at each vertex of the polygon, Gouraud shading then interpolates these light values while the polygon is being drawn to arrive at a separate light intensity for each pixel. After arriving at the light intensity for the pixel being drawn, this light intensity is combined with the color of the polygon, using a lighting table. The final, lighted color is then drawn.

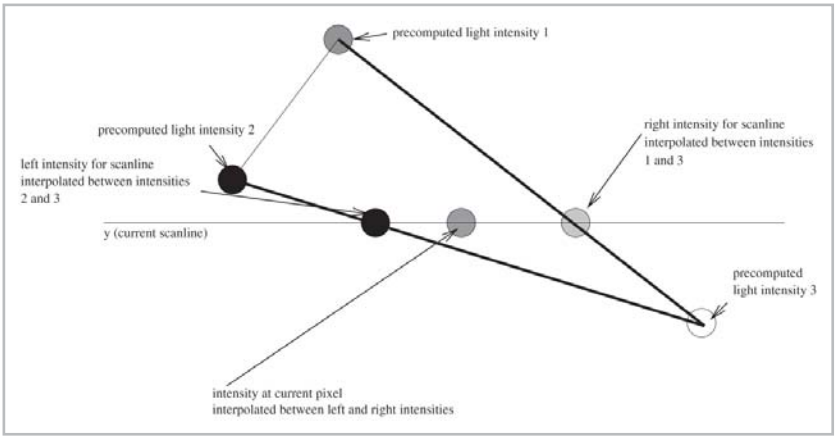


Figure 2-15: Gouraud shading computes light intensity at each vertex, then interpolates these precomputed intensities as the polygon is being drawn, giving a unique light intensity to every pixel.

The effect of Gouraud shading is to smooth out over the entire polygon the changes in light intensity which occur among the polygon's vertices. Also, since light intensities are computed at the vertices of the polygon, and since all polygons in an object share vertices, the light changes over the entire 3D object are also smoothed out, resulting in an overall smooth appearance of the surface. See Figure 2-16.



Figure 2-16: Gouraud shading makes a surface appear smooth and round.

Note that Gouraud shading is not always desirable, because it "washes out" all the sharp changes in light intensity which occur over the entire 3D object. If the object being modeled is supposed to appear round, then this is fine. But, if the object is supposed to have some sharp corners or edges, then the light intensity should indeed change suddenly from one face to the next because of the physical properties of the surface. But simple Gouraud shading would smooth over all sharp changes in light intensity, including the desired ones.

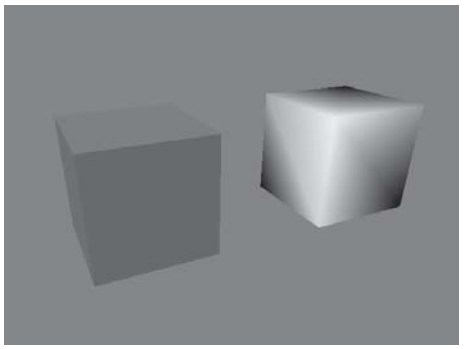


Figure 2-17: Gouraud shading eliminates all sharp edges, even if these edges should actually appear.

Solving this problem would require storing vertex normals per polygon instead of per object; this would effectively allow each shared vertex to have several vertex normals, one for each face touching the vertex. All vertex normals for one vertex would be the same if the surface is supposed to appear rounded at that point; the vertex normals would be different if an edge should appear between the adjacent faces.

A popular technique for allowing smooth edged polygons and sharp edged polygons in the same model is to use *smoothing groups*. With this technique, we assign a 32-bit long integer to each polygon, which is then treated as 32 Boolean flags. During the creation of a 3D model, the 3D artist can turn each of these on or off. When determining each vertex normal for a polygon, the long value is subjected to a logical AND operation with the long value of each neighbor. If the result is not zero, the neighbor's normal is included in the calculation; if it is zero, that neighbor is excluded. This means that each vertex can have more than one vertex normal depending on which polygon it is being considered for. By carefully setting the bits, the artist can specify which polygons affect which adjacent polygons, and thereby control smoothing (or lack thereof) on each edge.

Phong Shading

Phong shading takes Gouraud shading one step further. With Gouraud shading, we evaluated the lighting equation once per vertex, then we interpolated the precomputed intensities while drawing the polygon, giving an interpolated light value per pixel.

With Phong shading, we actually re-evaluate the entire lighting equation at each pixel. To do this, we again use vertex normals, and interpolate the normal vector itself while we are drawing the polygon. We use the interpolated normal vector in the lighting equation, and recompute the light for each pixel. Naturally, this requires a large amount of computation for each pixel.

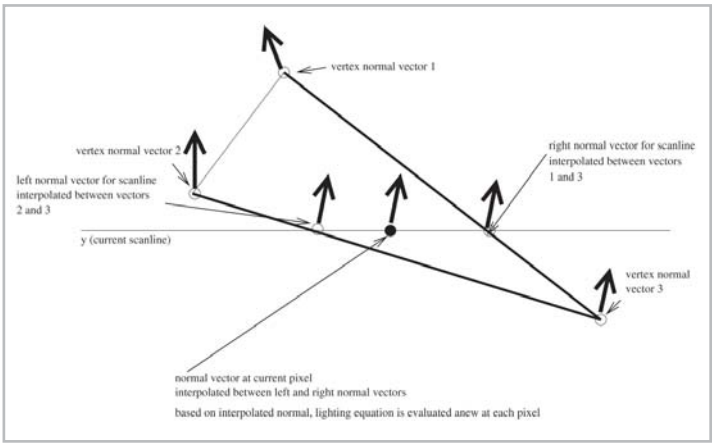


Figure 2-18: Phong shading interpolates the vertex normals themselves while the polygon is being drawn, and re-evaluates the entire lighting equation for each pixel using the interpolated normal vector.

Light Maps

A *light map* is a special form of a texture map (covered in the next section). Essentially, the idea is to associate a square grid, called the light map, with each polygon. The entire light map covers the whole polygon, and thus each element in the light map covers a certain small area of the polygon. We call each element in the light map a *lumel*, short for luminance element. To compute the light intensities, we evaluate the lighting equation once for each lumel, meaning that we evaluate the lighting equation several times for one polygon. The finer the grid, the more times the light equation needs to be evaluated, but also the greater the quality of the final result.

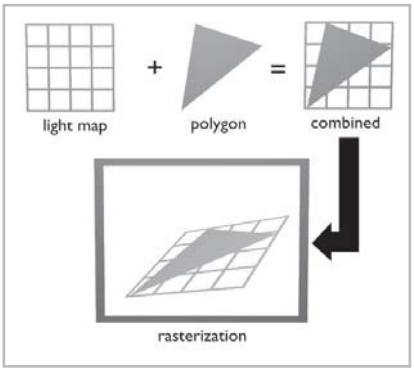


Figure 2-19: Light mapping.

Although we evaluate the lighting equation at several points on the polygon (at each lumel), this is only a one-time preprocessing step (assuming static and not dynamic lighting). After computing and saving the intensities in the light map, we simply store the precomputed light map with the polygon. Then, when drawing the polygon, for each pixel we determine the corresponding lumel in the light map (a non-trivial computation, as we see in the next section), combine the intensity in the light map with the polygon's color by using the lighting table, and finally draw the lighted pixel to the screen.

Light mapping can be seen as being somewhere between Gouraud and Phong shading. With Gouraud shading, we only computed the light at the vertices of the polygon; with Phong shading, we computed light at every single pixel. With light mapping, we define an arbitrary grid of lumels on top of the polygon, and compute the light intensities at every grid element.

The most difficult part of light mapping is the determination, during the rasterization of a particular polygon, of which lumel in the light map corresponds to the current pixel on-screen which we are drawing. When rasterizing a polygon, we know that all pixels we draw do correspond to (i.e., result from the perspective projection of) some point on the polygon and thus to some point in the light map, but we do not immediately know which point exactly. If you have a super-human memory, you may recall that we briefly mentioned this problem in passing in the introductory companion book *Linux 3D Graphics Programming*. To summarize, it is not exactly easy or intuitive to calculate which lumel on the polygon corresponds to the current pixel of the polygon being drawn, but it is possible, as we will see in grueling detail in the next section. This calculation is actually one particular application of the technique called *texture mapping*, which allows us to associate arbitrary images with a polygon. Therefore, let us now turn to the topic of texture mapping, returning to light mapping again afterwards.

Texture Mapping

As recently as the mid-1990s, an interactive 3D graphics application with texture mapping was quite a sensation for personal computers. Today, end users expect texture mapping even in simple 3D graphics applications. Therefore, it is important to understand this technique.

The goal of texture mapping is to draw the pixels of a polygon with colors coming from a separate 2D image. (The image can also be 3D, but for now assume the image is 2D.) The image is called the *texture*; the individual pixels within the texture image are often called *texels*. Seen this way, the next question is: given a pixel of a polygon on-screen, which texel from the texture should be drawn?

The answer to this question requires us to map the image onto the polygon. For instance, we might want to draw the pixels of the polygon so that it appears that the image were "glued on" to the surface of the polygon. This is by far the most common application of texture mapping. To "glue" an image onto a polygon, we need to consider the coordinate space of the polygon, and we also need to define a new coordinate space to contain the image. This new coordinate space containing the image is *texture space*, which we look at in more detail in the next section.

Team-Fly®

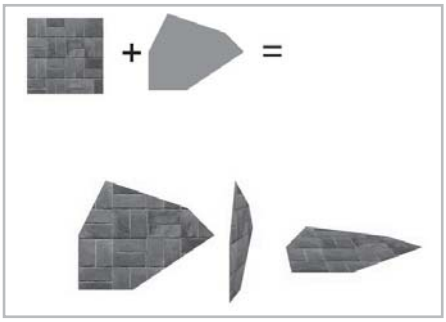


Figure 2-20: A texture, a non-texture-mapped polygon, and the same polygon with the texture map applied, in several different orientations. Notice that regardless of the orientation of the polygon, the texture map appears to be glued onto the surface of the polygon.

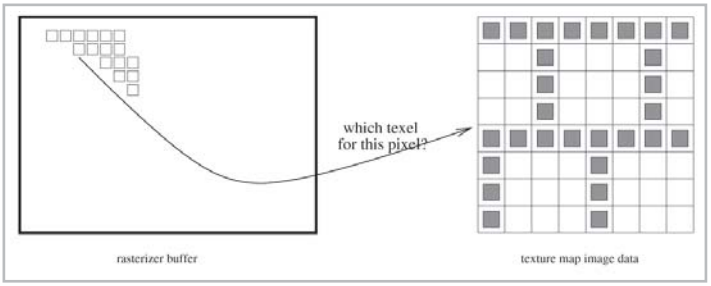


Figure 2-21: The fundamental question of texture mapping. Given a pixel coordinate of a polygon we are drawing, which pixel from the texture image should we draw?

Texture mapping requires us to perform the following steps:

1. Define a texture. The texture is simply a 2D image from which the colors of the polygon will be taken, pixel by pixel, during rasterization.
2. Define a texture space. This is a coordinate system to specify the location, orientation, and size of the image in 3D world coordinates.
3. Define a mapping between texture space and the world space coordinates of the polygon's vertices.
4. Reverse project screen coordinates into texture coordinates during rasterization. This means we must determine, based on a 2D (x,y) pixel coordinate obtained while rasterizing a polygon, the corresponding coordinates (u,v) in texture space, given the texture space of step 2 and world-to-texture-space mapping of step 3.
5. Map the computed texture space coordinates to integer (u',v') coordinates in the 2D texture image, and draw the pixel using the color at the (u',v') coordinates in the texture image.

This is a correct but somewhat slow means of performing texture mapping. First of all, let's look at each step to make sure we understand the basic procedure. Then, we'll look at an optimized version of the texture mapping process, based on the u/z, v/z, and $1/z$ terms. Based on this optimized procedure, we'll develop the C++ classes necessary to implement texture mapping.

Step 1: Define a Texture

The step which requires the least mathematical trickery is the first step: defining the texture itself.

Storing Texture Data

The class `l3d_texture` defines a texture in the l3d library. It inherits a texture orientation from class `l3d_texture_space`, which is covered in the next section. It also manages an object actually containing the texture image data, stored in variable `tex_data` and of type `l3d_texture_data`. The member variable `owns_tex_data` specifies whether this texture object actually "owns" the texture data or not. Several textures, with different orientations or sizes, can share the same texture data to save memory, but only one texture can actually own the data. The owner of the data is the one responsible for freeing the memory of the texture data when it is no longer needed.

The class `l3d_texture_data` actually stores the texture data. One `l3d_texture_data` object may be referenced by several `l3d_texture` objects. Member variable `id` is an integer variable used to uniquely identify the texture data. We'll need this identifier when we write Mesa routines to draw texture-mapped polygons. The member variables `width` and `height` are the width and height of the texture data, in pixels. Note that the width and height must be powers of two, solely for performance reasons: performing an arithmetic modulo operation (the operator % in the C language) can be implemented as a bit-wise logical AND operation when using powers of two, which is faster than the general purpose modulo operator. For instance, the expression 2335%256 in C can be rewritten as 2335&255, because 256 is a power of two, but 2335%17 cannot be rewritten as a bit-wise logical AND operation because 17 is not a power of two.

The member variable `data` is a linear array of `unsigned char`. It contains the image data and is of size width height bytes-per-pixel. The image data is stored linearly in left-to-right and top-to-bottom order within the `data` array. Each pixel in the array is stored in the same format as required by the screen. For example, if a pixel on-screen requires four bytes, one pixel in the texture is also stored using four bytes. Storing the textures in the native screen format means that the rasterizer can directly copy pixels from the texture into the buffer without needing to do any pixel format translation.

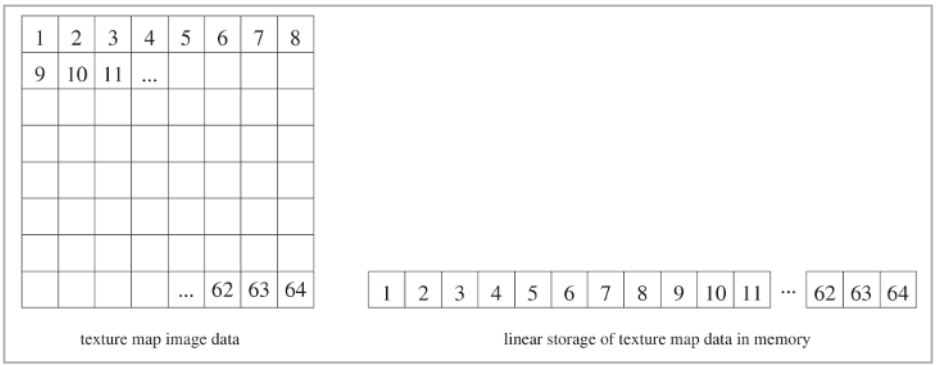


Figure 2-22: The storage in memory of the texture data: left to right and top to bottom. The textures are internally stored in the same pixel format as required by the screen; this means that each texel block may represent more than one byte.

Listing 2-4 presents the texture classes discussed so far. It also declares two classes, `l3d_texture_space` and `l3d_texture_computer`, which we cover in the coming sections.

Listing 2-4: `texture.h`

```
#ifndef __TEXTURE_H
#define __TEXTURE_H
#include "../../tool_os/memman.h"

#include <stdlib.h>
#include <stdio.h>
#include "../vertex/coord.h"

class l3d_texture_data {
  public:
    int id;
    l3d_texture_data(void) {id=0; data = NULL; }
    virtual ~l3d_texture_data(void) {if(data) delete [] data; }
    int width, height;
    unsigned char *data;

};

class l3d_texture_space {
  public:
    l3d_coordinate O, U, V;
};

class l3d_texture : public l3d_texture_space
{
  public:
    bool owns_tex_data;
    l3d_texture_data *tex_data;
    l3d_texture(void) {owns_tex_data = false; }
    ~l3d_texture(void) {if (owns_tex_data) delete tex_data; }
};

class l3d_texture_computer {

  private:
    l3d_matrix _world_to_tex_matrix;
  public:
    static const l3d_real EPSILON_VECTOR = float_to_l3d_real(0.0001);
    l3d_point O; l3d_vector U, V;
    l3d_real *fovx,*fovy;
    int *screen_xsize, *screen_ysize;

    l3d_real u, v;

    const l3d_matrix& world_to_tex_matrix(const l3d_texture_space &tex_space) {
      l3d_vector
      U_vector = tex_space.U.original - tex_space.O.original,
                 V_vector = tex_space.V.original - tex_space.O.original,
                            U_cross_V_vector = cross(U_vector, V_vector);

      l3d_vector w = U_cross_V_vector;
      l3d_vector v_x_u = cross(V_vector,U_vector),
                         u_x_w = cross(U_vector, w),
                                 w_x_v = cross(w, V_vector),
                                         o_vec(tex_space.O.original.X_,
```

```
                                              tex_space.O.original.Y_,
                                              tex_space.O.original.Z_,
                                              int_to_l3d_real(0));

        _world_to_tex_matrix.set(
          w_x_v.X_, w_x_v.Y_, w_x_v.Z_, dot(o_vec,w_x_v)*int_to_l3d_real(-1),
          u_x_w.X_, u_x_w.Y_, u_x_w.Z_, dot(o_vec,u_x_w)*int_to_l3d_real(-1),
          v_x_u.X_, v_x_u.Y_, v_x_u.Z_, dot(o_vec,v_x_u)*int_to_l3d_real(-1),
          int_to_l3d_real(0),
          int_to_l3d_real(0),
          int_to_l3d_real(0),
          dot(U_vector,w_x_v));

        return _world_to_tex_matrix;
      }

};

#endif
```

Classes for Loading Textures from Disk

Now we know how to store texture data in memory, but what about creating the texture image data in the first place? Typically, textures are created with a bitmap paint program, such as the GIMP program under Linux. The textures should be created with dimensions that are powers of two, for instance, 64 by 64 pixels. The textures need to be saved to disk in a file format which we can then read in and place into our `l3d_texture_data` class.

The l3d classes for loading textures support the portable pixmap (PPM) file format developed by Jef Poskanzer. Converters exist (under Linux and other operating systems) which convert most graphics formats to and from PPM format, and the better bitmap graphics programs also directly support loading and saving PPM files.

The main reason for using PPM files is simplicity: parsing PPM files is extremely easy. The bytes in the file consist of an identifying header, a width specification, a height specification, a maximum color value, and then the pixel image data in left-to-right and top-to-bottom order, with three decimal values for each pixel corresponding to red, green, and blue intensities for the pixel. All fields are separated by white space (blanks, tabs, carriage returns, or line feeds). The exact specifications for the PPM format are in the online manual under the keyword "ppm."

Class `l3d_texture_loader` is an abstract class that loads a texture from disk into memory. See Listing 2-5. The constructor requires a parameter of type `l3d_screen_info`, because the texture loader needs to know the pixel format of the screen in order to convert the texture data, during the loading process, into the pixel format needed by the screen. The virtual method `load` loads the image file stored in the given filename. The width and height of the image file are read from the file by the texture loader and are stored in member variables `width` and `height`. Member variable `data` then holds the actual image data from the image file, in the proper pixel format for the screen. This data then eventually gets assigned to an `l3d_texture_data` object, where it can then be referenced by an `l3d_texture` object.

Listing 2-5: `texload.h`

```
#ifndef __TEXLOAD_H
#define __TEXLOAD_H
```

```
#include "../../tool_os/memman.h"

#include "../../tool_2d/scrinfo.h"
#include <stdio.h>

class l3d_texture_loader {
  protected:
    l3d_screen_info *si;
  public:
    l3d_texture_loader(l3d_screen_info *si) {
      this->si = si;
      data = NULL;
    }
    virtual ~l3d_texture_loader(void) {};

    virtual void load(const char *fname) = 0;
    unsigned char *data;
    int width, height;
};

#endif
```

Class `l3d_texture_loader_ppm` is a texture loader specialized for loading PPM images from disk. See Listings 2-6 and 2-7. The overridden `load` method loads a PPM file by calling the protected routine `process_tok`. The `process_tok` method is a finite state machine which moves from state to state. In a particular state, there are only a certain fixed number of valid inputs. Each input causes a certain action to occur and a change to another state. By studying the manual page for the PPM file format, it should be quite easy to follow the logic of the file parsing code. The most important part of the code takes place in states 6 and 15 (for ASCII or binary format PPM files, respectively). Here, we convert the RGB value we just read from the PPM file into the native color format needed by the screen, by using the `ext_to_native` method of the `l3d_screen_info` object. Remember that `ext_to_native` also conveniently automatically allocates a new palette entry if we are using a palette color model.

Listing 2-6: `tl_ppm.h`

```
#ifndef __TL_PPM_H
#define __TL_PPM_H
#include "../../tool_os/memman.h"

#include "texload.h"

class l3d_texture_loader_ppm : public l3d_texture_loader {
  protected:
    int process_tok(char *tok);
    unsigned char *cur_pdata;
    int state;
    int cur_r, cur_g, cur_b;

  public:
    l3d_texture_loader_ppm(l3d_screen_info *si) :
    l3d_texture_loader(si) {};
    void load(const char *fname);
};

#endif
```

Listing 2-7: tl_ppm.cc

```
#include "tl_ppm.h"
#include "../../system/sys_dep.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../../tool_os/memman.h"

int l3d_texture_loader_ppm::process_tok(char *tok) {
  switch(state) {
    case 0: {
        if(strcmp(tok,"P3")==0) {
          state=1;
          return 1;
        }
        else if (strcmp(tok,"P6")==0) {
          state=10;
          return 1;
        }
        else return 0;
      }

    case 1: {
        sscanf(tok,"%d",&width);
        state=2;
        return 1;
      }

    case 2: {
        sscanf(tok,"%d",&height);
        data = new unsigned char [width*height*si->bytes_per_pixel];

        cur_pdata = data;

        state = 3;
        return 1;
      }

    case 3: {
        int maxcol=0;
        sscanf(tok,"%d",&maxcol);
        si->ext_max_red = si->ext_max_green = si->ext_max_blue = maxcol;
        state = 4;
        return 1;
      }

    case 4: {
        sscanf(tok,"%d",&cur_r);
        state = 5;
        return 1;
      }

    case 5: {
        sscanf(tok,"%d",&cur_g);
        state = 6;
        return 1;
      }

    case 6: {
```

```
    int blue=0;
    sscanf(tok,"%d",&cur_b);

    unsigned long native_col = si->ext_to_native(cur_r, cur_g, cur_b);

    register int i;
    unsigned long mask = MAX_BYTE;
    char shift = 0;

    for(i=0; i<si->bytes_per_pixel; i++) {
      *cur_pdata++ = (native_col & mask) >> shift;
      mask <<= BITS_PER_BYTE;
      shift += BITS_PER_BYTE;
    }

    state = 4;
    return 1;
  }

case 10: {
    sscanf(tok,"%d",&width);
    state = 11;
    return 1;
  }

case 11: {
    sscanf(tok,"%d",&height);
    data = new unsigned char [width*height*si->bytes_per_pixel] ;

    cur_pdata = data;

    state = 12;
    return 1;
  }

case 12: {
    int maxcol=0;
    sscanf(tok,"%d",&maxcol);
    si->ext_max_red = si->ext_max_green = si->ext_max_blue = maxcol;
    state = 13;
    return 2;
  }

case 13: {
    unsigned char c;
    sscanf(tok,"%c",&c);
    cur_r = c;
    state = 14;
    return 1;
  }

case 14: {
    unsigned char c;
    sscanf(tok,"%c",&c);
    cur_g = c;
    state = 15;
    return 1;
  }

case 15: {
```

```
        unsigned char c;
        sscanf(tok,"%c",&c);
        cur_b = c;

        unsigned long native_col = si->ext_to_native(cur_r, cur_g, cur_b);

        register int i;
        unsigned long mask = MAX_BYTE;
        char shift = 0;

        for(i=0; i<si->bytes_per_pixel; i++) {
          *cur_pdata++ = (native_col & mask) >> shift;
          mask <<= BITS_PER_BYTE;
          shift += BITS_PER_BYTE;
        }

        state = 13;
        return 1;
      }

    default: fprintf(stderr,"unknown error parsing ppm file");
      return -1;
  }
}

void l3d_texture_loader_ppm::load(const char *fname) {
  FILE *fp;
  char *tok;
  char s[256]="";
  int result;

  fp=fopen(fname, "r");
  if(!fp) {
    fprintf(stderr,"Error opening texture ppm file %s", fname);
    return;
  }

  state = 0;
  fgets(s, 256, fp);
  while (!feof(fp)) {
    tok = strtok(s," ");
    while(tok) {
      if(tok[0]=='#') break;

      result = process_tok(tok);
      tok = strtok(NULL," ");
    }
    if (result==2) break;
    fgets(s, 256, fp);
  }

  if(!feof(fp) && result==2) { //- P6 binary ppm file
    s[1] = 0;
    s[0] = fgetc(fp);

    //- skip comment lines before begin of binary image data
    while (s[0] == '#') {
      s[0] = fgetc(fp);
      while(s[0] != '') {
        s[0] = fgetc(fp);
```

```
      }
    }

    //- process binary image data
    while(!feof(fp)) {
      process_tok(s);
      s[0] = fgetc(fp);
    }
  }
}
```

Practical Issues in Dealing with Texture Image Files

You may have some existing image data that you wish to convert into a PPM file for use as a texture. The PPM utilities, available for many operating systems and included on the CD-ROM, offer several command-line utilities for converting to and from various file formats. You can find these by typing **man -k ppm** and **man -k pnm**. (PNM stands for portable anymap, meaning either a bitmap with two colors or a pixmap with arbitrary colors.) Some of the more important utilities are `giftopnm` and `djpeg -pnm`., which convert GIF or JPEG files to PPMs. Don't forget that the dimensions of the texture should be powers of two.

Another issue arises when using an indexed or palette color model. A single PPM file can have more colors than the maximum number of entries in the palette, which would cause palette overflow in the call to `ext_to_native`. A further problem is that we usually don't have just one texture, but instead have several texture files, each with many colors. Even if the palette is big enough to hold the colors for one texture, the total number of colors in all texture files is probably greater than the maximum palette size. If we are working with a palette, it is therefore essential to generate a *common palette* for all textures. Conceptually, this works by examining all colors in all input files and finding a palette with colors that most closely match most of the colors in most of the textures—a perfect match for all colors in all textures is, of course, impossible. The program `ppmquantall` generates a common palette of a specified size for a given set of input images, maps the original colors in the images to the new images in the common palette, and overwrites the original input files. For instance, to generate a common palette consisting of 128 colors for all PPM files in the current directory, you would type **ppmquantall 128 *.ppm**. Remember, this overwrites the original input files, so make sure you have backups before you execute this command. Also, you might want to artificially make the common palette size smaller than the actual palette size. By doing this, you slightly reduce the overall color quality for all textures, but also leave a certain number of entries in the actual palette free for later use. Then, when we are computing the lighting and fog tables, we have some custom colors we can allocate in case no existing color can be found that closely matches the desired light or fog level. If you generate a common palette that uses up all of the entries in the actual palette, then you have no entries left over to compensate for poor matches when computing the lighting and fog tables. This may or may not be a problem. If the original common palette contains a wide variety of colors and a wide variety of intensities, then maybe there will be a lot of good color matches when computing the lighting and fog tables. The sad fact of life with a palette is that you quite simply have a limited number of colors available for use; either you risk poor lighting and fog tables if you want the best possible

texture quality, or you sacrifice color resolution in the textures in exchange for more freedom when computing the lighting and fog tables.

There are also a number of other useful PPM utilities for performing image processing operations on image files—smoothing, scaling, rotating, brightening, and so forth. (See the online manual.) Also, the GIMP bitmap editing program under Linux offers many powerful image manipulation functions in a graphical interface, allowing you to immediately preview and try out a number of visual effects.

Step 2: Define a Texture Space

Having defined a 2D texture image, we need to somehow position this texture image in our 3D world. This is because the 2D texture image is supposed to affect the colors of the 3D polygons comprising our virtual world; it only makes sense, then, that the texture image must also be positioned within the 3D world.

Positioning the texture image in our 3D world first requires us to define a *texture space* containing the texture image. Texture space is simply a normal coordinate system with three axes and an origin. We name the three texture space axes the *u* axis, *v* axis, and *w* axis, corresponding to the *x*, *y*, and *z* axes in our normal left-handed world coordinate system.



CAUTION Do not confuse the w axis, the third dimension of texture space, with the homogeneous w coordinate. These two terms have nothing to do with one another!

The special thing about texture space is that we associate a color with each point in the space. Thus, you can think of texture space as a solid, colored, infinitely large cube, where each point in the volume of the solid cube may be assigned a separate color. The convention we define for texture space is that the 2D texture image we defined in step 1 is located in the 2D square in texture space defined by the corner points (0,0,0), (0,1,0), (1,1,0), (1,0,0). Thus, the texture image occupies exactly one square unit, starting at the origin, and located in the *uv* plane of texture space. Outside of this single square unit, but still in the *uv* plane, we can think of the texture as being duplicated, meaning that each square unit in texture space, starting and ending on whole numbers, contains the colors of the 2D texture image. As for the *w* axis, for 2D textures we impose the restriction that the colors in texture space do not depend on *w*; they only depend on the (*u*,*v*) values. (*3D textures* also allow for color variation along the *w* axis.) Graphically, you can imagine that the 2D *uv* plane is simply duplicated infinitely up and down the entire length of the *w* axis.

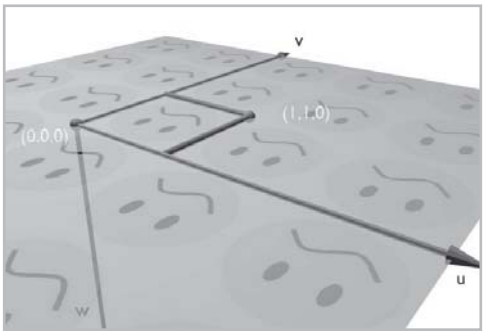


Figure 2-23: Texture space is a normal 3D space, with the extension that each point contains a color. The 2D texture image occupies the square area in texture coordinate points located from (0,0,0) to (1,1,0). Outside of this square area, the texture image is typically repeated.

Team-Fly®

It is important to understand that a coordinate in texture space (also often called a (u,v) coordinate) is essentially equivalent to a pixel in the texture image. Texture space is a colored space; the texture image defines the colors in the space. Therefore, each point in the texture space corresponds to a pixel in the texture image. Here are some examples: the (0,0,0) coordinate in texture space corresponds to the first pixel in the texture image; (0.5, 0.5, 0) is the pixel in the middle of the image; (0.999, 0.999, 0) is the last pixel in the image; and (1,1,0) is again the first pixel in the image, because the texture repeats itself.

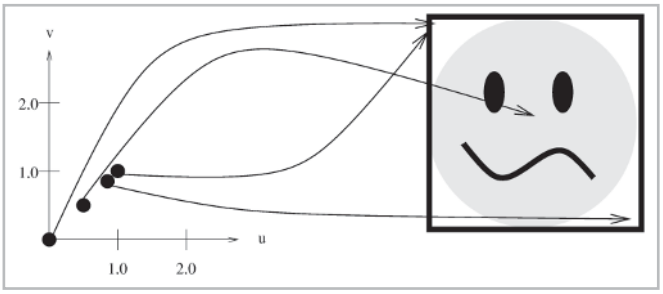


Figure 2-24: Points in texture space correspond to colors in the texture image.

Since texture space is a normal 3D space, we can define it in terms of three vectors, defining the axes, and an origin point. We name the origin point *O*, and the three axis vectors *U*, *V*, and *W*. Since we have said that for 2D textures, which is all we use in this book, the *w* texture coordinate does not affect the color in texture space, it turns out that we rarely need the *W* axis. When we do need the *W* axis vector, we can compute it as $U \times V$.



NOTE Computing the W vector as $U \times V$ has one disadvantage: the size of the resulting vector is then arbitrary. We could simply normalize the W vector after the cross product, but the problem is that the sizes of the texture space axis vectors define the scaling of the texture in world space. We look at this more closely in the next section. This means that by computing the W vector dynamically, we lose the freedom to specify a custom scaling of the texture along the w axis. Since for 2D textures, we ignore the w axis anyway, this is unimportant. But with true 3D textures, where the w axis may be important, we should define our own W axis to have full control over the entire specification of the 3D texture space.

This means that to store a texture space, we just need to store *O*, *U*, and *V*. Instead of storing the vectors *U* and *V*, we instead store the tip points of these vectors; the base point of these vectors is then *O*. As usual, we subtract base from tip to obtain the vector. We store the tip points to allow for consistent transformation using the `l3d_coordinate` class, as mentioned earlier in this chapter in the discussion about surface normal vectors.

Class `l3d_texture_space`, presented earlier in Listing 2-4, stores a texture space in this manner. It simply has three member variables, all of type `l3d_coordinate`: `O`, `U`, and `V`. Class `l3d_texture`, as we saw earlier, inherits from `l3d_texture_space`. This means that a texture also has a texture space associated with it.

Step 3: Map Between Texture Space and World Space

Defining a texture space in terms of a texture coordinate system actually has the effect of positioning, orienting, and scaling the texture within 3D space. Recall that texture space is a colored space. Our colored 2D texture image is located in the *uv* plane of texture space, in the unit square going from (0,0,0) to (1,1,0). By positioning the origin point of the texture coordinate system, we are also effectively positioning the first pixel of the texture image. By orienting the axes of the texture coordinate system, we are also effectively orienting the texture image. By scaling the axes of the texture coordinate system, we scale the size of the texture image.

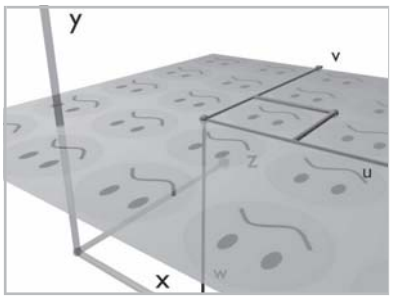


Figure 2-25: Changing the position of the origin of the texture coordinate system has the effect of positioning the texture image.

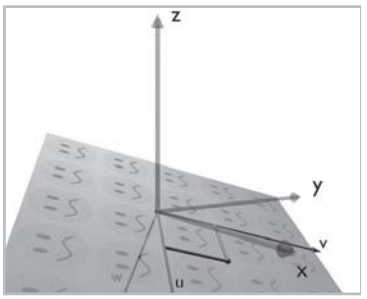


Figure 2-26: Changing the orientation of the texture space axes has the effect of orienting the texture image.

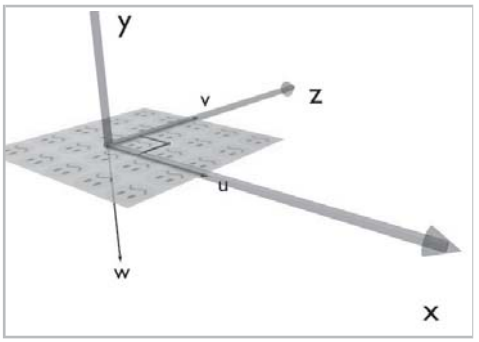


Figure 2-27: Changing the size of the texture coordinate axes has the effect of scaling the texture image.

The purpose of the texture coordinate system, then, is to define the position, orientation, and scaling of the texture image within the 3D world coordinate system.

Given a texture space, we should then define a mapping between texture space and world space. This is nothing more than a change in coordinate system. This means that given a particular coordinate in texture space, we should be able to find out the coordinate in world space. Conversely, and more commonly, we must also be able to determine, given a coordinate in world space, what its coordinates are in texture space.

Let's now look at each of these mappings separately.

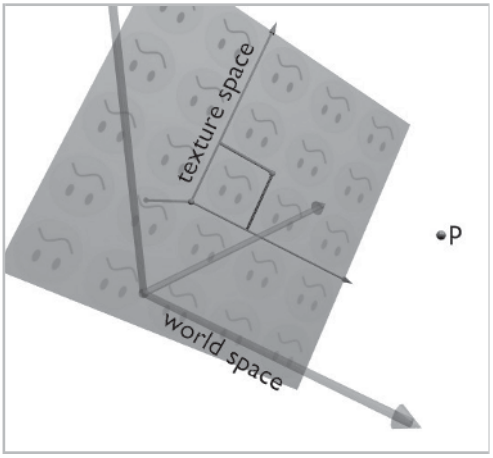


Figure 2-28: Mapping between texture coordinates and world coordinates. If we know the coordinates of a point P in one of the coordinate systems, we should be able to find the coordinates of the point in the other coordinate system.

The mapping from texture space to world space is very easy to compute. Recalling the relationship between matrices and coordinate systems, we can simply write the texture coordinate system as a matrix, using the first three columns as the axis vectors and the last column as the origin point. This matrix, when multiplied with a column vector representing a location in texture space, then converts the texture space coordinates to world space coordinates, for the reasons discussed in the introductory companion book *Linux 3D Graphics Programming*.

Equation 2-18

$$\begin{bmatrix} U_x & V_x & W_x & O_x \\ U_y & V_y & W_y & O_y \\ U_z & V_z & W_z & O_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The mapping from world space to texture space is more difficult, but it is the mapping which we more often need. Why do we need it more often? The reason is that the final texture mapping strategy we implement needs texture coordinates for each world space polygon vertex. The idea is that we ultimately wish to associate texture coordinates—and thus colors from the texture image—with every pixel we draw on-screen for a particular polygon. There are two ways of attacking the problem. The first is the direct *pixel-level* approach, covered in the next section, where we explicitly compute a texture coordinate for every pixel as it is being drawn. The second is a *vertex-level* approach, covered in the section titled "An Optimized Texture Mapping Strategy," where we only explicitly compute texture coordinates for each *world space vertex* of a polygon, and interpolate these precomputed values to arrive at the texture coordinates for each pixel. For the second, vertex-level approach, we know the world space coordinates of each polygon vertex, and need to compute the texture coordinates for each vertex. In other words, given the world space coordinates of a vertex, what are its coordinates in texture space?

So, how do we compute the matrix that maps from world space to texture space? Conceptually, the answer is simple: invert the matrix that converts texture space to world space. It is tempting, but incorrect, to use the same strategy we used (in the introductory companion book *Linux 3D Graphics Programming*) for deriving the camera transformation matrix. In that case, we

simply took the transpose of the matrix to find its inverse. But recall why this works: the vectors for camera space are all mutually orthogonal, and are all of unit size. While our texture space vectors are probably orthogonal, they are not always of unit size. This is because, as we said earlier, we must allow for different-sized texture space axis vectors to allow for scaling of the texture. Therefore, since the *U*, *V*, and *W* vectors are not of unit size, we cannot invert the matrix simply by taking its transpose.

In this case, we need to perform a bit of tedious algebra to invert the matrix. We need to express the matrix as a set of simultaneous equations, solve these equations algebraically, then recast the expression into matrix form. We are going to go through every step of the algebra in excruciating detail because it is important that you understand how to find the solution to such mathematical problems, which arise often in 3D graphics. But don't worry—we are going to use the computer to do most of the algebra for us; specifically, we are going to use the symbolic algebra package Calc, which is an extension to the Emacs editor. Calc is included on the CD-ROM. The important thing is to understand what the problem is, how to cast it in a form that can be solved by Calc, and how to use Calc.

Our problem is to invert a matrix. First of all, let's rewrite the problem as a set of linear equations. Then, we'll see how we can use Calc to solve the set of linear equations.

The matrix presented earlier converts from texture space to world space. This means that multiplying texture space coordinates (*u*,*v*,*w*) with this matrix gives us world space coordinates (*x*,*y*,*z*). Mathematically, this can be expressed as:

Equation 2-19

$$\begin{bmatrix} U_x & V_x & W_x & O_x \\ U_y & V_y & W_y & O_y \\ U_z & V_z & W_z & O_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Performing the matrix multiplication yields the following system of equations.

Equation 2-20

$$\begin{aligned} x &= u\,U_x + v\,V_x + w\,W_x + O_x \\ y &= u\,U_y + v\,V_y + w\,W_y + O_y \\ z &= u\,U_z + v\,V_z + w\,W_z + O_z \end{aligned}$$

This system of linear equations is merely a different representation for the same information contained within the matrix equation. Given *u*, *v*, and *z* in texture space, the equations tell us how to find *x*, *y*, and *z* in world space.

We now want to solve this system of equations for *u*, *v*, and *w*, based on known values of *x*, *y*, and *z*. This takes us from world space to texture space. Solving this system of linear equations is exactly the same as inverting the equivalent matrix.

The system of equations is solvable; there are three equations and three unknowns. You may wish to try to solve the system of equations by hand, using the normal techniques of grade school algebra. If your patience level is the same as mine, you will notice that this system is rather tedious to solve. Scores of terms need to be copied from one step to the next, some terms cancel out surprisingly, and others look like they will cancel out but actually do not. The danger of making a tiny sign error somewhere is great, leading to incorrect results and wasted hours of scribbling, not to

mention a cramped hand. Most of the work involved in solving this system of equations seems tedious, repetitive, manual, and error-prone. It seems like a perfect job for the computer. The Calc package can help us here. So, let's now take a detour to look at how to use the Calc package to solve this system of equations.



NOTE If you are uninterested in understanding how to solve the system of equations, you can skip this section and just use the results. However, I sincerely hope that you take the time to learn to use Calc and to understand how to solve such problems using Calc as a tool. Sooner or later, every 3D programmer will be confronted with solving systems of equations. A technique as ubiquitous as texture mapping cannot be fully understood without understanding the solutions to the equations.

Calc: A Symbolic Algebra Package

Calc is a free (zero cost and open source) mathematical tool which runs as part of the Emacs environment. It can perform numeric and symbolic mathematical manipulations. Its ability to perform *numeric* math operations means that Calc can compute quantities specified as numbers, just as with a normal desk calculator—for instance, (37–140)/15.3. Its ability to perform *symbolic* math operations means that Calc can solve systems of equations specified algebraically in terms of known and unknown variables. For the problem at hand—converting from world space to texture space—we are mostly interested in the symbolic capabilities of Calc.

Here is an example of Calc's symbolic capabilities. If we have the system of equations:

Equation 2-21
$$\begin{aligned} x + y &= k \\ x - y &= 3k \end{aligned}$$

then Calc can automatically solve this system of equations for x and y to give us the solution:

Equation 2-22
$$\begin{aligned} x &= 2k \\ y &= -k \end{aligned}$$

Internally, therefore, Calc has performed the typical grade school algebraic manipulations for solving systems of equations. For this example, the equations are very easy to solve by hand, but in the case of inverting the texture space to world space matrix, the system of equations is quite simply tedious to solve by hand. The symbolic algebra capabilities of Calc relieve us of exactly this burden. Let's first take a brief tour of how to use Calc, then focus on using the symbolic algebra capabilities to solve the texture mapping equations.



NOTE Calc has excellent online info documentation, including many hands-on tutorials. Alternatively, within Calc, press ? and type i to display the info documentation from within Calc. Since we don't have enough space here to describe all of Calc's features—many of which are very interesting and useful for solving the types of problems which arise in 3D graphics—I encourage you to have a look at Calc's info documentation.

Starting and Exiting Calc

To use Calc, you must have both Emacs and the Calc package installed, and you must know how to do basic editing in Emacs. See the Appendix for instructions on installing Calc.

With Emacs and Calc installed, you can get started as follows.

1. Type **emacs** and press **Enter**. The Emacs window appears.
2. In the Emacs window, press **Alt+x** to run an extended command. Notice the text "M–x" which appears at the bottom of the screen. Type **calc** and press **Enter**. Two new small windows appear at the bottom part of the existing Emacs window. See Figure 2-29. Notice that the cursor is now located within the leftmost Calc window.
3. Press **Ctrl+x, 1**. This maximizes the Calc window to occupy the entire Emacs window.



Figure 2-29: Calc immediately after starting. The two windows at the bottom are the Calc windows.

To exit Calc, simply kill the Calc buffer. When the cursor is in the Calc window, press **Ctrl+x, Ctrl+k, Enter**.

Next, let's enter some simple expressions and let Calc evaluate them.

Stack-Based Computation

Fundamentally, Calc operates with a stack of mathematical expressions. The typical flow of operations in Calc is to enter entries on the stack, then to perform operations with the entries on the stack. Some operations, such as negation, operate only on the topmost element of the stack. Other operations, such as addition, operate on the top two elements of the stack, removing these and replacing them with one new element, the result of the operation. This style of operation is also called *reverse Polish notation* or RPN; some desk calculators operate in this mode.

Let's now perform some basic computations to see the stack in action and to learn some typical Calc commands.

1. Press **'**, a single quote character. This tells Calc that you are about to input an algebraic expression, and that this expression should be placed on top of the stack. Notice the prompt "Algebraic:" which appears at the bottom of the screen.
2. Type **125** and press **Enter**. Notice the text "1: 125" which appears within the Calc window. This means that entry number 1 of the stack has just been set to value 125.
3. Type **n**. This negates the top entry on the stack. Notice the number 125 changes to –125.
4. Press **'**, type **25**, and press **Enter**. This enters a new entry on the top of the stack. The existing entries on the stack are pushed down by one position. Notice that the entry number next to the number –125 has changed to 2, indicating that it is now entry number 2 on the stack. Notice the new line "1: 25" which appears, indicating that the top entry of the stack is the number 25.
5. Press +. This adds the top two entries on the stack and replaces them with a new entry on the top of the stack representing the result. Notice that the entries –125 and 25 on the stack disappear and are replaced by a new entry with a value of –100, the result of adding –125 and 25.
6. Press **Shift+u** to undo the last operation. Notice that the original two entries reappear.
7. Press / to divide the second entry on the stack (–125) by the first entry (25). Notice that the result (–5) appears on top of the stack.
8. Press **Shift+u** to undo the last operation.
9. Press **Tab**. This exchanges the top two entries on the stack.
10. Press / to divide the second entry on the stack (25) by the first entry (–125). Notice that the result (0.2) is different than before, because the order of the stack entries was different; we have exchanged dividend and divisor.
11. Press **Shift+u** to undo the last operation.
12. Press **Space**. Notice that the top entry of the stack is duplicated. Press **Space** three more times to duplicate the top entry three more times.
13. Press **Backspace**. Notice that the top entry of the stack disappears. Press **Backspace** three more times to delete the next three entries from the top of the stack.
14. Press **Ctrl+u**. This allows you to enter a numerical argument to an upcoming command. Type **2** as a numerical argument. Now, enter the command to be executed with the argument of 2. Press **Space**, the "duplicate entry" command.
15. Notice that the top two entries of the stack have been duplicated, because of the numerical argument 2 which we entered earlier.

Now we know how to enter items onto the stack, how to duplicate and delete them, and how to perform basic arithmetic operations on them. Here is a summary of the most important Calc commands:

- Press **'** (single quote) to enter an algebraic expression.
- Press +, –, /, or ***** to add, subtract, divide, or multiply the top two entries of the stack, replacing them with a new entry containing the result.
- Type **n** to negate the top entry on the stack.
- Press **Space** to duplicate the entry on top of the stack.

- Press **Ctrl+u**, type number *n*, and press **Space** to duplicate the top *n* entries on the stack.
- Press **Tab** to exchange the top two entries on the stack.
- Press **Shift+u** to undo operations.
- Press **Backspace** to delete the entry on top of the stack.

Entering and Editing Mathematical Entities

We've now seen how to use Calc to manipulate numbers. Calc's real attraction for us, though, is its symbolic algebra facility. We can enter variables in Calc simply by using alphanumeric identifiers in our expressions. Let's try entering some symbolic equations for Calc to solve.

1. Press **'** to allow algebraic entry.
2. Type **x + y = 10** and press **Enter**. Notice that the formula, with variables, appears on the Calc stack.
3. Press **'**, type **x – y = 5**, and press **Enter**. This second formula appears on the stack.
4. Press **Ctrl+u** and type **2**. This enters an argument of two for the next command to come.
5. Type **v**, then press **Shift+p** to pack the top two entries on the stack into a vector of equations. Notice that the two equations are placed into one list of equations, separated by commas and surrounded by square brackets.



NOTE It would also have been possible to enter the vector of equations directly in one step, instead of entering two separate equations then packing them. To enter both equations in one step, simply separate the equations with a comma and place square brackets around the entire list of equations during the entry of the expression.

6. Type **a**, then press **Shift+s** to solve the system of equations on the top of the stack. Notice the prompt that appears at the bottom of the screen, "Variable to solve for:"
7. Type **[x,y]** and press **Enter**. This tells Calc that we want to solve the system of equations for two variables, *x* and *y*. Do not forget to type the square brackets.
8. Notice that Calc solves the system and displays the solution x=7.5, y=2.5.

Note that when entering expressions containing terms that should be multiplied, you can either explicitly specify the multiplication with the * (asterisk) character, as in "a*b", or you can simply separate the terms with a space, as in "a b". You should not write the terms directly next to one another, as in "ab", because in this case, Calc interprets this as one term instead of two terms multiplied together.

Although we now know how to use Calc to solve systems of equations, we should not try just yet to solve the texture mapping equations. Instead, we should continue to look at some other editing features of Calc. Later, when we actually solve the texture mapping equations, we will be confronted face to face with the full complexity of the solutions, and will need all tools at our disposal to reduce the solutions to the simplest form possible.

The following list summarizes the features of Calc that we use in the following sections to solve and simplify the texture mapping equations.

- Editing expressions. Calc allows you to edit the top expression on the stack. This is extremely useful for performing manual simplifications or rearranging an existing formula so that Calc can more easily recognize and simplify expressions. Edit the top expression on the stack as follows. Press **'**, a single straight quote character. A new window, with the title "Calc Edit Mode," opens, containing the top expression of the stack. Use the normal Emacs editing commands to edit the text of the expression. Press **Ctrl+c** twice to accept your changes, or **Alt+#**, **x** to cancel your changes.
- Display modes. Calc offers different ways of displaying algebraic expressions. The Normal mode is a compact, straight-line representation, where two variables next to one another are understood to be multiplied together, as with standard algebraic notation. The Big mode is an easy-to-read ASCII representation, where fractions are displayed vertically instead of horizontally. The C mode is a representation using the syntax and operators of C language expressions. The TeX mode is a representation suitable for inclusion in the typesetting program TeX, which is often used in academic circles for publishing documents with large amounts of math. (The equations in this book were all typeset using the LyX front end to the LaTeX macro extension to the TeX program.) Change the display mode in Calc as follows. Type **d**. Then, for Normal mode, press **Shift+n**. For Big mode, press **Shift+b**. For C mode, press **Shift+c**. For Tex Mode, press **Shift+t**.

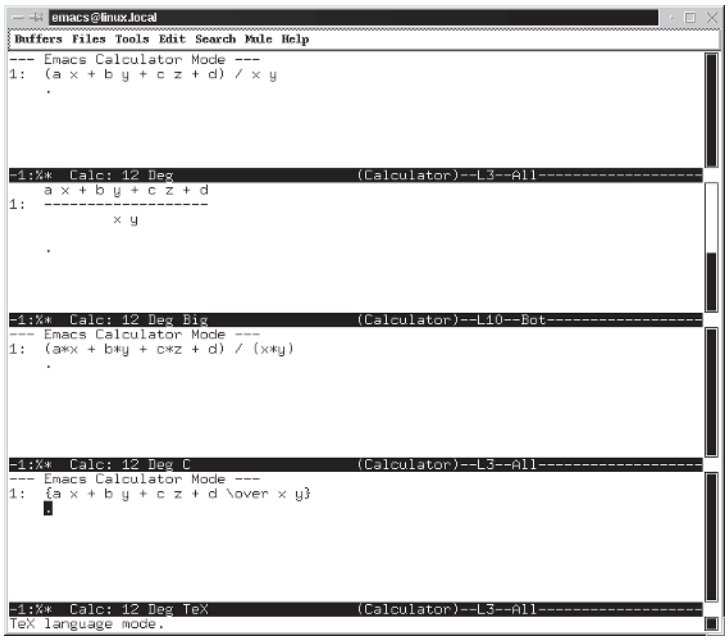


Figure 2-30: Normal, Big, C, and TeX display modes in Calc.

- Vector entry. Calc has facilities for dealing with row and column vectors of numbers or variables. To enter a row vector, first press **'** to allow algebraic entry. Then, press **[** to begin the vector. Enter each element of the vector, separated by the comma (**,**) character. Press **]** to end

the vector, and press **Enter** to finish the input. To enter a column vector, follow the same procedure, but separate the elements of the vector with the semicolon (**;**) character.

- Matrix entry. Calc also allows entry and computation with matrices of numbers or variables. To enter a matrix, begin as if you were entering a vector: press **'** then **[**. Enter the first row of elements in the matrix, separating the elements of the matrix with the comma (**,**) character. To finish a row, enter a semicolon (**;**) character, and continue typing the elements of the next row. Continue until the matrix is complete. Then press **]** and **Enter**.

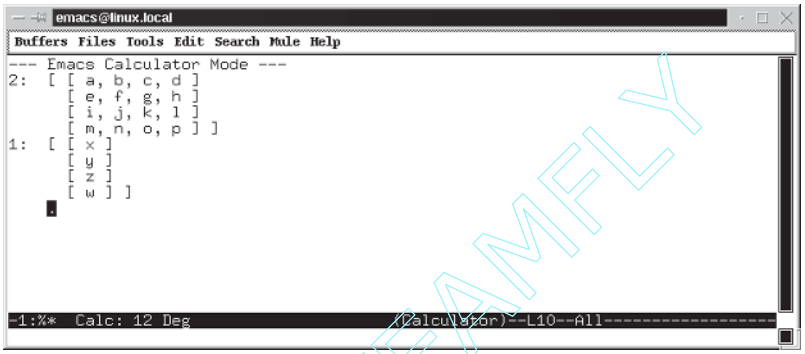


Figure 2-31: A matrix and a column vector entered within Calc. The matrix was entered with the key sequence '[a,b,c,d;e,f,g,h;i,j,k,l;m,n,o,p] followed by Enter. The vector was entered with the key sequence '[x;y;z;w] followed by Enter.

TEAMFLY

- Simplifying equations. Calc offers many commands to simplify algebraic expressions. The most important commands are the simplify and normalize-rational commands. The simplify command tries to simplify the expression, using rules which ordinarily might be too slow to apply. For instance, simplify combines non-adjacent sums of terms: $a+b+2a$ is simplified to $b+3a$. The normalize-rational command attempts to rearrange a formula to be the quotient of two polynomials. For instance, $a+b/c+d/e$ is simplified to $(ace+be+cd)/(ce)$. Invoke these commands as follows. Type **a** to begin entering an algebraic command. Then, to apply the simplify command, type **S**; to apply the normalize-rational command, type **n**. The simplification occurs on the topmost element of the stack.
- Collecting terms. The collect command of Calc rearranges a formula as a polynomial in a given variable. For instance, if we have the expression $ax+bx+cx$, then collecting the x terms yields the expression $(a+b+c)x$. Invoke the collect command as follows. Type **a** to begin entering an algebraic command, then type **c**. Notice the prompt "Collect terms involving:" which appears at the bottom of the screen. Type the name of the term which should be collected, and press **Enter**.
- Selections. Often, it is useful to operate on just one small part of an expression, rather than the expression as a whole. Calc allows us to define a *selection*, which is simply a smaller portion of an expression. After defining a selection, most Calc commands then operate only on the selection. Most commonly, after a selection you will apply one of the simplification commands (**a, S** or **a, n**) or the edit command (**'**, the straight quote character). This allows you to simplify or edit a smaller portion of the entire expression. Apply a selection within Calc as follows. Move the cursor to be on top of the beginning of the subexpression that you want to select. Type **j** to enter selection mode. Then, type one of the following keys to define a

Team-Fly®

selection: **s** for the smallest subexpression at the cursor position, **m** for more of the current subexpression, **n** for the next subexpression after the current selection, or **c** to cancel selection mode and select the entire formula. After defining a selection, notice that the rest of the equation is replaced with dot characters to indicate that these parts of the expression are temporarily inactive.

Now that we know some of the more important tools that Calc offers for editing and simplifying expressions, let's move on to the actual practical reality of solving systems of equations within Calc.

Solving Systems of Equations

Recall, we want to use Calc to solve a system of equations. The last hurdle standing in our way is an unfortunate quirk of Calc's method of solving systems of equations. The quirk is that when solving a system of equations for many variables, only one variable—usually the last variable solved for—tends to have a reasonably simple form. The other variables tend to have incredibly complicated, non-simplified forms, spanning ten lines or more and defying human comprehension.

For instance, say we have a system of three equations with three unknown variables *u*, *v*, and *w*, and we ask Calc to solve this system by using the **a**, **S** command described earlier. But when entering which variables to solve for, we can enter them in any order: we can enter **[u,v,w]**, **[v,w,u]**, or **[w,u,v]** (among other permutations). In the first case, variable *w* would typically have the simplest form; in the second case, *u*; in the third case, *v*.

This may sound unimportant, but in practice it is very important. You might think that the complicated expressions for the non-last variables could be simplified with the simplify and normalize-rational commands described in the previous section. Unfortunately, this is not the case. The problem is that many common subexpressions differ only by a sign—for instance, $a/(b–c) + a(c–b)$. In this case, the subexpression $(c–b)$ is actually the same as the denominator $(b–c)$ of the first expression, only negated. When this same situation of identical-but-negated subexpressions is expanded to expressions containing ten or more terms, Calc is not always able to recognize the commonality among the subexpressions, and is thus often not able to simplify the expressions further. If you notice such expressions, it is possible for you to edit the offending expression manually, negating both numerator and denominator, so that both terms then have truly identical denominators, which would make it explicitly clear to Calc that the expression can be simplified. However, with extremely long expressions, even the human capability for pattern recognition can begin to reach its limits.

Therefore, the best way to solve systems of equations in Calc is to solve the system repeatedly, each time allowing a different variable to be last in the solution vector, and therefore the simplest. Assuming our system has the three unknown variables *u*, *v*, and *w*, we would proceed as follows.

1. Enter the system of equations to be solved, within square brackets and separated by commas.
2. Duplicate the system of equations before solving it: press **Space**. In this way, you still have a copy of the original equations, even after solving it (which replaces the original system of equations with its solution).

3. Solve the duplicated system with one particular variable, say *u*, as the last one. Type **a**, **S**, type **[v,w,u]** and press **Enter**.
4. The solution for all three variables appears, but only the solution for the last variable *u* is sufficiently simple to be understandable. Edit the solution: press **'** (straight quote). Delete everything before and after the last solution. Press **Ctrl+c** twice to accept your changes. Now, only the solution for the last variable remains on top of the stack.
5. Press **Tab** to exchange the top two entries of the stack. The system of equations now appears on top of the stack, and the solution for one variable beneath it.
6. Repeat again starting at step 2, but with a different variable, say *v*, as the last one. Continue until all variables have been solved for as the last variable in the solution vector.

This procedure gives you the best results from the equation solver. But often, even if a variable was solved for as the last variable in the solution vector, the resulting equation can still be simplified further with the simplify and/or normalize-rational commands. In particular, it is often helpful to simplify only parts of the equations at a time. Do this with the selection command. Select a reasonably small subexpression, consisting of, say, four to five terms. First simplify the expression by typing **a, S**, then apply the normalize-rational command by typing **a, n**. Then, select more of the expression with **j, m**, and continue to simplify. If Calc ever fails to simplify, simply undo the operation, select a smaller subexpression, and try again. If even that fails, and you are convinced that the solution can be simplified further, you need to help Calc out by making common terms easier to recognize. Most often, common-but-negated terms will appear scattered throughout the expression. By editing the expression and judiciously negating terms so that no mathematical change in value is effected (for instance, by negating both numerator and denominator of a fractional expression), it is often possible to make similar-but-negated terms exactly identical. After doing this, try simplifying again; Calc will then usually recognize the common terms which you made explicit, and simplify further.

To summarize, when solving systems of equations in Calc, you should solve the system multiple times, each time with a different variable as the last one in the solution vector. Use the duplicate command to avoid needing to enter the system of equations multiple times. Simplify the resulting solutions a piece at a time by using selections, the simplify command, and the normalize-rational command.

Solving the Texture Mapping Equations with Calc

Now, at long last, we have enough information to solve the texture mapping equations in Calc. We solve the system with the procedure described in the previous section: duplicate, solve with one variable as the last variable, repeat for other variables. Simplify each solution piece by piece by using selections.

Immediately after entering the system of equations from Equation 2-20 into Calc, the Calc window appears as in Figure 2-32.

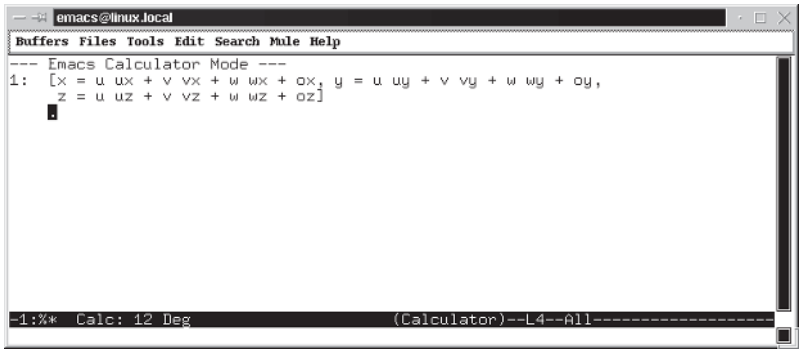


Figure 2-32: The texture space to world space equations in Calc.

After solving the system with **a, S** followed by **[v,w,u]**, the Calc window appears as in Figure 2-33. Notice the complexity of the solutions for *v* and *w*, and the relative simplicity of the solution for *u*.

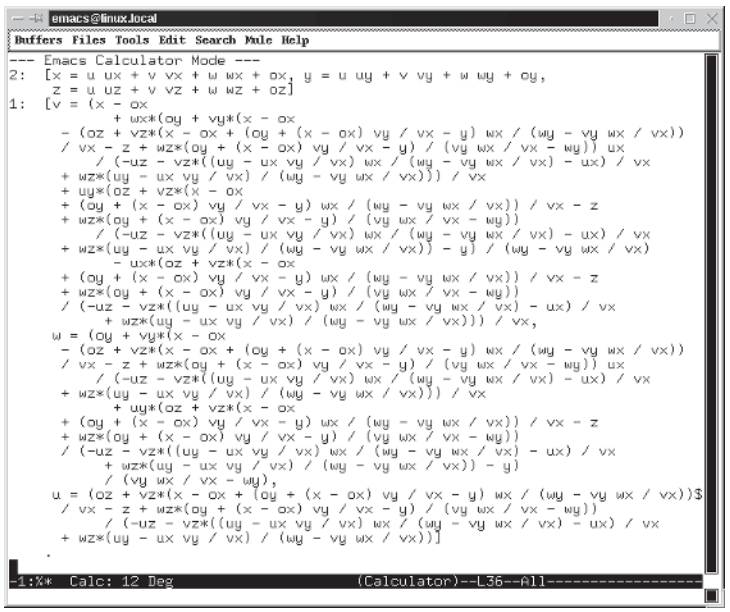


Figure 2-33: The system solved for [v,w,u].

We then edit the expression to remove everything but the solution for the last variable, *u*. After doing this, the Calc window appears as in Figure 2-34.

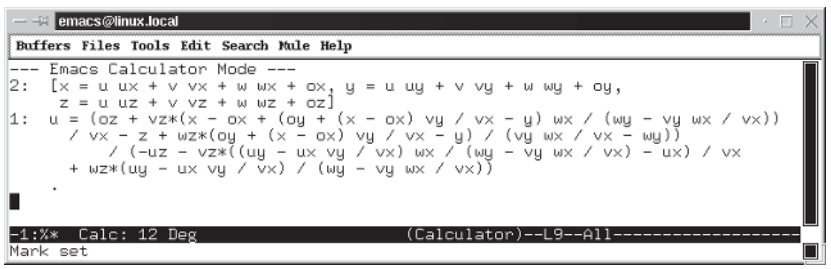


Figure 2-34: The solution for u has been isolated.

We can simplify the solution for *u* further. A naive attempt to apply the simplify and normalize-rational commands to the entire expression at once results in Calc reporting a "division by 0" error. But the expression can be simplified. By simplifying and normalizing the equation piece for piece using selections, and by collecting terms for *x*, *y*, *z*, *ux*, *uy*, and *uz*, the expression finally simplifies to the form shown in Figure 2-35. I encourage you to try to arrive at the same result yourself by using Calc.

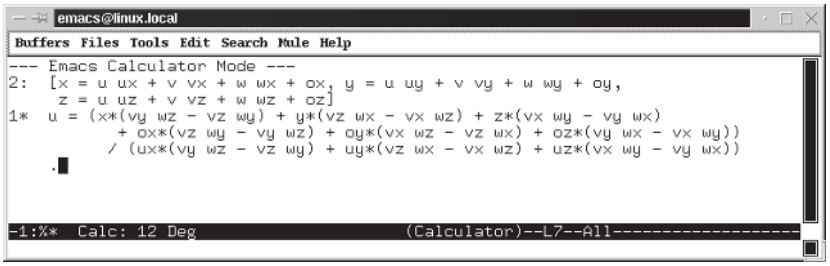


Figure 2-35: The solution for u has been simplified.

By repeating the solution procedure for variables *v* and *w*, we arrive at the following simplified solutions for *u*, *v*, and *w*.

Equation 2-23

$$u = \frac{x(v_z w_y - v_y w_z) + y(v_x w_z - v_z w_x) + z(v_y w_x - v_x w_y) + o_x(v_y w_z - v_z w_y) + o_y(v_z w_x - v_x w_z) + o_z(v_x w_y - v_y w_x)}{u_x(v_z w_y - v_y w_z) + u_y(v_x w_z - v_z w_x) + u_z(w_x v_y - w_y v_x)}$$

$$v = \frac{x(u_y w_z - u_z w_y) + y(u_z w_x - u_x w_z) + z(u_x w_y - u_y w_x) + o_x(u_z w_y - u_y w_z) + o_y(u_x w_z - u_z w_x) + o_z(u_y w_x - u_x w_y)}{u_x(v_z w_y - v_y w_z) + u_y(v_x w_z - v_z w_x) + u_z(w_x v_y - w_y v_x)}$$

$$w = \frac{x(u_z v_y - u_y v_z) + y(u_x v_z - u_z v_x) + z(u_y v_x - u_x v_y) + o_x(u_y v_z - u_z v_y) + o_y(u_z v_x - u_x v_z) + o_z(u_x v_y - u_y v_x)}{u_x(v_z w_y - v_y w_z) + u_y(v_x w_z - v_z w_x) + u_z(w_x v_y - w_y v_x)}$$

These equations are the solution to the world-to-texture-space problem. Given variables *x*, *y*, and *z* representing world space coordinates, and a texture space definition as specified by the *O* point and vectors *U*, *V*, and *W*, these equations give us the equivalent coordinates *u*, *v*, and *w* in texture space. As you can see, the solution is not completely intuitive, and would have been rather tedious to derive by hand.

There is still more structure to be discovered in these equations. Notice the recurring pairs of differences which appear: (*vz wy – vy wz*), (*vx wz – vz wx*), and so forth. These terms are actually components of cross product vectors. In particular, the solution for *u* contains the components of the cross product vector $V \times W$; the solution for *v* contains the components of $U \times W$; the solution for *w* contains the components of $V \times U$. The denominator in each case is $U \bullet (V \times W)$. Making this simplification, and recasting the equation into matrix form, gives us Equation 2-24. The subscripts on each cross product expression mean to take the corresponding component of the vector after the cross product operation. For instance, $(V \times W)_x$ means to take the cross product of vectors *V* and *W*, and to use the *x* component of the resulting vector.

Equation 2-24

$$\begin{bmatrix} (W \times V)_x & (W \times V)_y & (W \times V)_z & -O \cdot (W \times V) \\ (U \times W)_x & (U \times W)_y & (U \times W)_z & -O \cdot (U \times W) \\ (V \times U)_x & (V \times U)_y & (V \times U)_z & -O \cdot (V \times U) \\ 0 & 0 & 0 & U \cdot (W \times V) \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ w_{tex} \\ w_{hom} \end{bmatrix}$$

Verify that this matrix form of the equation, using cross products, is the same as the Calc solution by performing the matrix multiplication, expanding the cross products, and dividing each component of the resulting solution vector (u, v, and w_{tex}) by the homogeneous component w_{hom}. We have subscripted the w texture component and the homogeneous w divisor with *tex* and *hom* to distinguish their separate functions; we use the homogeneous w_{hom} component to effect a division on all terms, since the solution equations for u, v, and w, as shown by Calc, all have a common denominator.

Therefore, the matrix in Equation 2-24 converts from world space coordinates to texture space coordinates; by multiplying a column vector in world coordinates with the matrix, we obtain a new column vector with the texture coordinates. Since this is a useful operation, we have encapsulated it into an l3d class: class `l3d_texture_computer`. Any class needing the ability to perform texture related computations inherits from this class. The method `world_to_tex_matrix` takes an object of type `l3d_texture_space` as a parameter, and returns the matrix, which converts coordinates from world coordinates into the texture space of the given `l3d_texture_space` object. The member variables `fovx`, `fovy`, `screen_xsize`, and `screen_ysize` are pointers to external integers specifying the horizontal and vertical screen size and field of view terms. We need these variables later within the Mesa 3D rasterizer to perform a reverse projection.

Remember that with 2D textures, we generally disregard the third texture coordinate. This means that when converting from world space to texture space, we can discard the third component of the resulting vector; only the first two components, u and v, are relevant for 2D image textures.

Now we know how to convert from texture space to world space, and from world space to texture space. This is an important step in texture mapping; with this understanding, the relationship between the texture space and world space is clearly defined, and thus the relationship between the texture image and the world space is clearly defined. The next step is to understand the relationship between screen space and texture space.

Step 4: Reverse Project from Screen Coordinates into Texture Coordinates

The practical goal of texture mapping is to determine, for each pixel of a polygon being drawn, which pixel from the texture image should be drawn. We can see this goal as being a conversion from screen coordinates to texture coordinates. We know the screen coordinates of each pixel we are drawing on-screen; what we need to know are the texture coordinates in texture space, which then give us the precise pixel coordinates within the texture image we need to display.

Given the previous section's matrix converting world coordinates to texture coordinates, it would seem logical to approach this problem by converting screen coordinates to world coordinates, then from world coordinates to texture coordinates. In theory, this should work fine. But, as far as I can tell, along that path lies madness, even with the help of a powerful symbolic algebra tool like Calc. So, let's assault the problem from a different angle—let's convert directly from screen space (pixel coordinates) back into texture space (texture coordinates), skipping the intermediate step of world coordinates. Our strategy is to start with texture space, then convert to world

space, then convert to screen space with a perspective projection. Then, we take the resulting system of equations, converting from texture space to screen space, and solve it for the texture space variables, thus resulting in equations converting from screen space to texture space. Again, we rely heavily on Calc's help to save us from the tedium of solving the equations by hand.

Let's start with the texture space to world space matrix, multiplied with a column vector representing a location in texture space. The result is a location in world space.

Equation 2-25
$$\begin{bmatrix} U_x & V_x & W_x & O_x \\ U_y & V_y & W_y & O_y \\ U_z & V_z & W_z & O_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Carrying out the matrix multiplication, we obtain:

Equation 2-26
$$\begin{bmatrix} uU_x + vV_x + wW_x + O_x \\ uU_y + vV_y + wW_y + O_y \\ uU_z + vV_z + wW_z + O_z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

We have said that for 2D texture images, we don't use the *w* texture component, because there is no color variation along the *W* axis in texture space. Therefore, the value of *w* does not affect the color we obtain from the texture image; any value of *w* can be used in the equation. The simplest value of *w* is 0, which causes the *w* term to conveniently disappear. After replacing *w* with 0, we obtain:

Equation 2-27
$$\begin{bmatrix} uU_x + vV_x + O_x \\ uU_y + vV_y + O_y \\ uU_z + vV_z + O_z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

We can rewrite this from matrix form into a system of linear equations.

Equation 2-28
$$\begin{aligned} x &= uU_x + vV_x + O_x \\ y &= uU_y + vV_y + O_y \\ z &= uU_z + vV_z + O_z \end{aligned}$$

Now, let's apply a simple perspective projection. We do this by dividing *x* and *y* by *z* to yield x_p and y_p (the subscript p stands for projected), resulting in Equation 2-29. This simple projection incorrectly ignores the effect of the field of view parameters and the reversal of the *y* axis orientation which actually occurs with the real perspective projection. We can use Equation 2-30 to convert from the real screen projected coordinates x_s and y_s, which include the field of view and *y* reversal, into the simple projected coordinates x_p and y_p. Since it is easy to convert between the real and the simple projected coordinates, and since the math is less tedious if we use the simple projected coordinates, the rest of the derivation for the texture mapping equations uses x_p and y_p. Any general observations about the variables x_p and y_p apply equally to the variables x_s and y_s.

Equation 2-29

$$\begin{aligned} x_p &= (uU_x + vV_x + O_x)/z \\ y_p &= (uU_y + vV_y + O_y)/z \\ z &= uU_z + vV_z + O_z \end{aligned}$$

Equation 2-30

$$\begin{aligned} x_p &= \frac{x_s - 0.5w_s}{(0.5\cot(0.5\theta)w_s)} \\ y_p &= \frac{0.5scale_y h_s - y_s}{(0.5\cot(0.5\theta)\frac{w_{Phys}}{h_{Phys}}h_s)} \end{aligned}$$

Let's now rearrange the first and second equations in Equation 2-29 to move the *z* to the left-hand side of the equations. This yields:

Equation 2-31

$$\begin{aligned} x_p\, z &= uU_x + vV_x + O_x \\ y_p\, z &= uU_y + vV_y + O_y \\ z &= uU_z + vV_z + O_z \end{aligned}$$

This is the system of equations we want to solve. First, let us consider what the equations in their current form tell us, before solving the equations for other variables. In their current form, this system of equations assumes we know the texture coordinates *u* and *v*. The equations then return three values: the screen *x* coordinate multiplied by the world *z* coordinate, the screen *y* coordinate multiplied by the world *z* coordinate, and the world *z* coordinate. Notice that the 3D world space *z* coordinate affects the first two equations. The reason is the perspective projection, which divides by *z*. The texture space to world space matrix, developed earlier, relates texture space to non-projected 3D world coordinates. Therefore, to relate texture space to projected 2D screen coordinates—which is the main goal, remember—we must "reverse project" the screen coordinates by multiplying them with the world space *z* coordinate. Thus, the 3D *z* coordinate, due to the perspective projection process, plays an important role in texture mapping.

Now that we understand what this system of equations tells us, let's solve the system for the variables we want to know. We wish to find the texture coordinates *u* and *v*. Since the 3D *z* coordinate also plays a role here, it is also an unknown. So we solve the system for variables *u*, *v*, and *z*. Do this by using Calc as described in the previous section, solving the system multiple times, each time with a different variable as the last variable in the solution vector. After solving the system in this way and simplifying the results, the final equations are as follows.

Equation 2-32

$$\begin{aligned} u &= \frac{x_p(O_yV_z - O_zV_y) + y_p(O_zV_x - O_xV_z) - O_yV_x + O_xV_y}{x_p(U_zV_y - U_yV_z) + y_p(U_xV_z - U_zV_x) + (U_yV_x - U_xV_y)} \\ v &= \frac{x_p(O_yU_z - O_zU_y) + y_p(O_zU_x - O_xU_z) - O_yU_x + O_xU_y}{x_p(U_zV_y - U_yV_z) + y_p(U_xV_z - U_zV_x) + (U_yV_x - U_xV_y)} \\ z &= \frac{O_x(U_zV_y - U_yV_z) + O_y(U_xV_z - U_zV_x) + O_z(U_yV_x - U_xV_y)}{x_p(U_zV_y - U_yV_z) + y_p(U_xV_z - U_zV_x) + (U_yV_x - U_xV_y)} \end{aligned}$$

This is an important equation for texture mapping. It tells us explicitly, for a given projected coordinate on-screen, the corresponding (*u*,*v*) coordinates in the texture space.

Notice that the *u*, *v*, and *z* variables are *non-linear* functions of the projected coordinates x_p, and y_p. Non-linear means that a change in x_p or y_p does not always result in the same amount of change in the *u*, *v*, or *z* variables. Mathematically, you can recognize this because the independent variables x_p and y_p appear in both the numerator and denominator of a fraction, meaning that a constant change in x_p or y_p leads to a non-constant change in the dependent variables *u*, *v*, or *z*. Changing the denominator of a fraction by a constant amount leads to a non-constant change in the final value; the difference between 1/2 and 1/3 is much greater than the difference between 1/10002 and 1/10003, though the denominator changes by a constant amount of one in both cases.

Geometrically, you can understand this non-linearity by examining Figure 2-36. When a polygon is tilted away from the projection plane—which is essentially our computer screen—a movement of one pixel on the screen corresponds to a non-constant movement on the polygon in 3D. This non-linearity is a fundamental property of the perspective projection process and causes (or indeed, is) the foreshortening which we observe in ordinary visual perception.

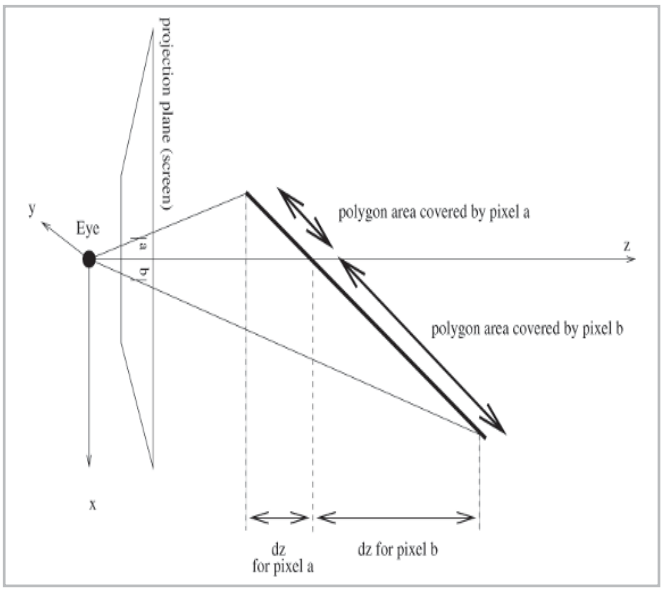


Figure 2-36: u, v, and z are not linear in screen space. Moving one pixel on-screen from a to b does not move you by a constant amount along the polygon, because of the perspective foreshortening.

A further observation about Equation 2-32 is that the numerator and denominator of each term, taken separately, are indeed linear with respect to x_p and y_p; it is the result, dividing numerator by denominator, which is non-linear.

Linearity of an equation with respect to x_p and y_p (and therefore, also with respect to x_s and y_s) is important because it allows us to compute the value at a particular screen location based on the value at a previous screen location. We used the same idea when developing the incremental algorithm for line drawing: the value for the next pixel could be calculated from the value of the current pixel. With regard to the texture mapping equations, this means that we can compute an initial value for the numerator and denominator based on the starting values of x_p and y_p for the first pixel of the texture mapped polygon we are drawing. Then, we can incrementally compute the numerator and denominator (requiring just one addition operation) for all other pixels in the polygon. We must perform the division, however, to obtain the u and v coordinates.

Step 5: Map Texture Coordinates to Integer Indices and Draw

After computing the (u,v) coordinates from the screen coordinates with the equation from the previous section, we map the real-valued (u,v) coordinates to integer indices within the 2D texture map image. The coordinates u and v each go from 0 to 1, with the convention that the texture

image exists within the square in texture space defined by corner points (0,0) and (1,1). Outside of the square (0,0) to (1,1)—that is, for *u* and *v* values greater than one or less than zero—we simply regard the texture as being duplicated.

The coordinates within the texture map image are integer coordinates from 0 to *width*–1 horizontally and from 0 to *height*–1 vertically, where *width* and *height* are both powers of two. So to map from the real-valued (*u*,*v*) to the indices within the texture map, we simply multiply *u* by *width* and *v* by *height*, yielding integer coordinates (*i*,*j*) within the texture map image. If *u* or *v* were greater than one or less than zero, this could lead to integer texture map indices lying outside of the texture image; to correct this and allow for repetition of the texture, we simply perform a logical bit-wise AND of *i* and *j* with *width*–1 and *height*–1. This has the effect of causing any values greater than *width* or *height* to "wrap around" back to zero again. Notice that this only works because *width* and *height* are powers of two; with powers of two, the AND operation is essentially a modulo operation, which gives us the value of the remainder after a division. Also, note that this wrap-around behavior helps compensate for the fact that pixels slightly outside of the polygon might be drawn due to numerical inaccuracy inherent to the DDA polygon drawing algorithm. If a pixel lies outside of the polygon, then a computed texture coordinate will lie outside of the texture image, but the wrap-around behavior allows us still to retrieve a valid texel from the texture. Instead of wrapping around, we could also clamp the values to force them to lie within the texture image.

Given the integer coordinates (*i*,*j*) within the texture image, we simply use the color at this location in the image (stored within the `l3d_texture_data` class) for drawing the current pixel of the texture mapped polygon being rasterized on-screen.

An Optimized Texture Mapping Strategy: u/z, v/z, 1/z

The previous sections described a pixel-level strategy for texture mapping. During rasterization of a polygon, we directly calculate from the screen coordinates the corresponding texture coordinates for each pixel.

An alternative strategy is a *vertex-level* approach, which we hinted at earlier. With a vertex-level approach, we only explicitly compute texture coordinates for the vertices of the polygon. We perform this computation in world space, by using the world space to texture space matrix developed earlier. Then, when drawing the polygon, we more or less interpolate the texture coordinates between the vertices. We say "more or less" because we actually do not interpolate the texture coordinates *u* and *v* themselves, but rather the related quantities *u*/*z* and *v*/*z*. Let's look at the texture mapping equations again to see where these *u*/*z* and *v*/*z* terms come from.

If we look at Equation 2-32, which converts projected coordinates into texture coordinates, we notice that the equations for *u*, *v*, and *z* all look similar. In particular, we can relate the equations for *u* and *v* with the equation for *z*. The denominator of z is the same as the denominator of *u* and *v*; therefore, *u* and *v* can be expressed as their numerators multiplied by *z*, and finally divided by the numerator of *z*. This is shown in Equation 2-33.

Equation 2-33
$$\begin{aligned} k &= O_x(U_zV_y - U_yV_z) + O_y(U_xV_z - U_zV_x) + O_z(U_yV_x - U_xV_y) \\ u &= \tfrac{1}{k}z(x_p(O_yV_z - O_zV_y) + y_p(O_zV_x - O_xV_z) - O_yV_x + O_xV_y) \\ v &= \tfrac{1}{k}z(x_p(O_yU_z - O_zU_y) + y_p(O_zU_x - O_xU_z) - O_yU_x + O_xU_y) \\ z &= \frac{k}{x_p(U_zV_y - U_yV_z) + y_p(U_xV_z - U_zV_x) + (U_yV_x - U_xV_y)} \end{aligned}$$

We can rearrange the equations as follows.

Equation 2-34
$$\begin{aligned} k &= O \cdot (V \times U) \\ \frac{u}{z} &= x_p\frac{O_yV_z - O_zV_y}{k} + y_p\frac{O_zV_x - O_xV_z}{k} + \frac{O_xV_y - O_yV_x}{k} \\ \frac{v}{z} &= x_p\frac{O_yU_z - O_zU_y}{k} + y_p\frac{O_zU_x - O_xU_z}{k} + \frac{O_xU_y - O_yU_x}{k} \\ \frac{1}{z} &= x_p\frac{U_zV_y - U_yV_z}{k} + y_p\frac{U_xV_z - U_zV_x}{k} + \frac{U_yV_x - U_xV_y}{k} \end{aligned}$$

In this form, we can see that the u/z, v/z, and $1/z$ terms are all linear functions of x_p and y_p. This means that calculating the next value of u/z, v/z, or $1/z$ for the next pixel location can be done with a simple addition to the value at the current pixel location. Furthermore, by dividing u/z by $1/z$, we arrive at the desired texture coordinate u; similarly, by dividing v/z by $1/z$, we arrive at v.

One relevant question is in which coordinate space z should be. Two obvious choices come to mind: the world space z coordinate or the camera space z coordinate. The best coordinate space to use for the z value is the camera space z coordinate. This is because the world space z coordinate can vary wildly; its values can be extremely large, extremely small, positive, or negative; the world z coordinate depends on the location of the object within world space, which can be completely arbitrary. On the other hand, the camera space z coordinate is a little more restrictive; it cannot be negative, has a minimum value (the near z clipping plane), and has a maximum value beyond which objects are too far from the camera to be seen. These additional restrictions on the possible values of the camera space z coordinate make it more amenable for use within the texture mapping computations.

The linearity of u/z, v/z, and $1/z$ suggests to us the following texture mapping strategy.

1. Define a texture and a texture space.
2. For each polygon, compute u, and v for each vertex index of the polygon, using the world space to texture space matrix. Store these u and v values in the new `tex_coord` member of the derived `l3d_polygon_ivertex_textured` class, covered in the next section.
3. After transforming polygons into camera space, also save the camera space z coordinate in the `tex_coord` member of `l3d_polygon_ivertex_textured`.
4. During rasterization, compute u/z, v/z, and $1/z$ for each vertex index of the polygon.
5. While stepping vertically between scanlines and horizontally between pixels, interpolate the u/z, v/z, and $1/z$ values. Do this by maintaining two sets of variables, one set for the left side and one set for the right side of the polygon. The left side of the polygon is always being drawn from a starting left vertex to an ending left vertex; the same for the right side of the polygon. Each vertical scanline step moves the current pixel from the starting vertex closer to the ending vertex. Thus, we also move from the starting u/z to the ending u/z; similarly for v/z and $1/z$. This means that for each scanline, we always have a current u/z, v/z, and $1/z$ value for the left side and the right side of the polygon. Then, for each pixel within the scanline, we

Team-Fly®

interpolate from the left u/z, v/z, and $1/z$ values to the right u/z, v/z, and $1/z$ values. This gives us a u/z, v/z, and $1/z$ value for every pixel of the polygon we rasterize. See Figure 2-37.

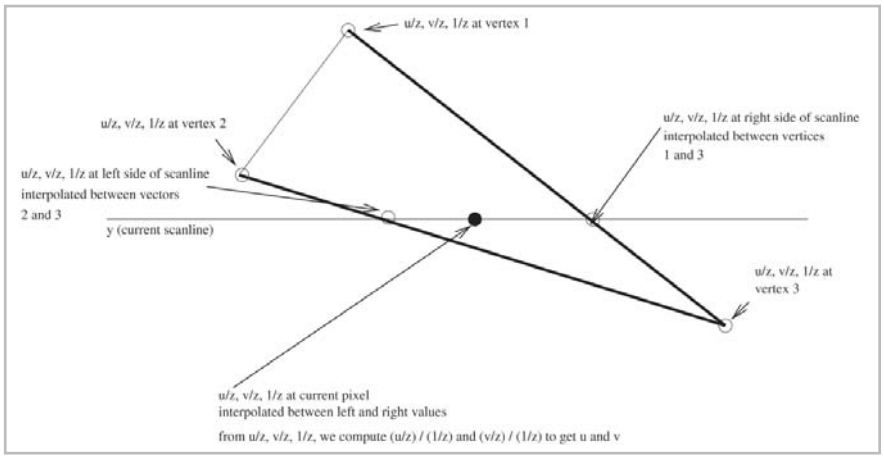


Figure 2-37: Interpolating u/z, v/z, and 1/z vertically and horizontally between vertices to arrive at a u/z, v/z, and 1/z value for each pixel.

6. Before drawing each pixel, divide the current u/z by $1/z$, and divide v/z by $1/z$. This returns the u and v values for the current pixel. Scale these to integer texture map coordinates, extract the appropriate pixel color from the texture map, and draw the current pixel with the color from the texture map.

The Division Operation and Texture Mapping

Notice that regardless of whether we use the pixel-level or vertex-level approach to texture mapping, we must perform a divide to arrive at the u and v values. With the pixel-level approach, we must divide the numerator by the denominator for the u and v terms; with the vertex-level approach, we must divide u/z and v/z by $1/z$. Thus, the pixel-level and the vertex-level approach are mathematically equivalent; it is just that with the vertex-level approach, we have made the relationship between u, v, and z more explicit.

The divide operation is time consuming. To perform texture mapping such that each pixel has the correct u and v values, we must perform at least one division per pixel. (We say one division and not two because we divide the u and v terms by the same denominator, meaning that we can compute the reciprocal of the denominator with one divide operation, then multiply the u and v terms with the precomputed reciprocal.)

It is not necessarily an absolute requirement that each and every pixel have the correct u and v values; as long as most of the values are correct or not too far from correct, the image will still likely be acceptable. Thus, one way of reducing the number of required divide operations, with a slight reduction in image correctness, is only to compute the true u and v values every so many pixels—say, every 16 pixels. Let's call each group of 16 horizontally adjacent pixels a *run* of pixels. Then, at the beginning and end of each run, the u and v values are correct, because we compute them as usual. For the pixels within a run, we linearly interpolate the u and v values between the start of the run and the end of the run. The interpolated u and v values will not be the correct u and v values as dictated by the texture mapping equations, because we cannot correctly interpolate u and

v in screen coordinates. However, they will not be "too far" from the original values, since we know that the endpoints of the run are correct. The smaller the runs, the less noticeable the error, but the larger the runs, the fewer divisions we must perform and the better the overall performance of the texture mapping code.

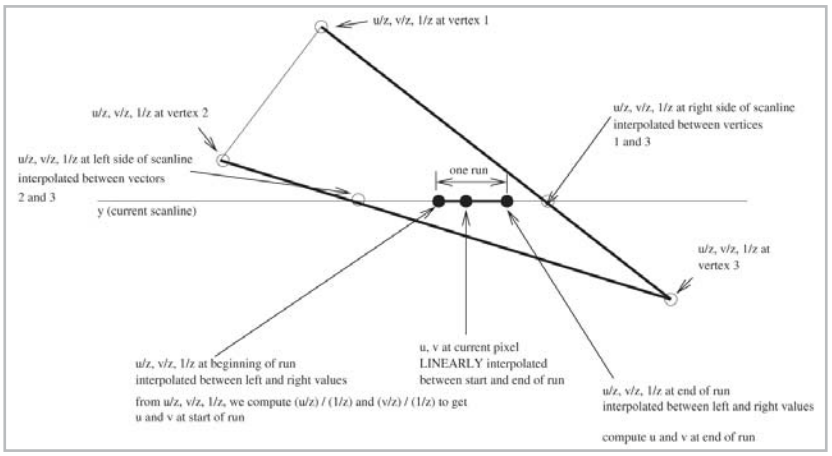


Figure 2-38: Dividing a horizontal row of pixels into runs. We only compute the true u and v values at the endpoints of the run. In between the endpoints, we linearly interpolate.

We've now covered the theory we need to perform texture mapping. The approach implemented in l3d is the vertex-level approach, interpolating *u*/*z*, *v*/*z*, and 1/*z*. Let's now look at the new polygon class needed to implement this texture mapping strategy.

Associating Textures with 3D Polygons

The class `l3d_polygon_3d_textured` is a texture mapped polygon. It is a 3D polygon with a texture and with texture coordinates stored with each vertex index. The class `l3d_polygon_ivertex_textured`, inheriting from `l3d_polygon_ivertex`, allows us to store the texture coordinates. It is an index to a polygon vertex with an additional member `tex_coord` to store a texture coordinate. Note that we store the texture coordinates with the vertex indices and not with the vertices themselves. The vertices themselves are shared among all polygons in an object, but the vertex indices belong to each individual polygon. Since the texture coordinates must be associated with each vertex of a polygon, and since adjacent polygons sharing a vertex might nevertheless have different texture coordinates at the same vertex, we must store the texture coordinates with the polygon's own vertex index and not with the shared vertex. The class `l3d_polygon_ivertex_textured_items_factory` is a factory returning items of the new type `l3d_polygon_ivertex_textured`; recall, the `l3d_list` class internally stores a factory for production of new list items during automatic expansion of the list. The factory overrides the `clone` method to copy itself; when copying a polygon, we should also copy the vertex index factory so that the copied polygon can produce the same kind of vertex indices (textured or non-textured) as the original polygon.

Returning to the class `l3d_polygon_3d_textured`, the normal constructor replaces the `iv_items_factory`, responsible for producing vertex index items for the (inherited) lists `ivertices`, `clip_ivertices`, and `temp_clip_ivertices`, with the new factory so

that the vertex index lists all now contain the extended vertex indices with texture coordinates. Therefore, all vertex index lists can store texture coordinates. Only after replacing the factory does the constructor then call the `init` method, which then creates the vertex index lists using the new factory. This is also why the empty base class constructors are called, to prevent the base class constructors from creating the lists before the new factory is in place. Class `l3d_polygon_3d_textured` also declares a copy constructor and a `clone` method, as usual.

The member variable `texture` stores the texture orientation and data.

The method `assign_tex_coords_from_tex_space` takes a texture space as a parameter, computes the texture coordinates based on the world space to texture space matrix, and stores the resulting texture coordinates with each vertex index in the `ivertices` list. Notice that we only use the first and second (*u* and *v*) components of the vector, since the third texture coordinate is unimportant for 2D textures.

The overridden methods `init_transformed` and `transform` perform the same transformation functions as in the parent class `l3d_polygon_3d`, but have been extended also to initialize and transform the `O`, `U`, and `V` member variables as well. This means that any transformation of the polygon also transforms the polygon's associated texture space, as well.

The overridden method `after_camera_transform` copies the current camera space *z* coordinate of each vertex into the corresponding vertex index object. We store the *z* coordinate in the `tex_coord.Z_` element of the vertex index. This means that the `tex_coord.X_` and `tex_coord.Y_` elements are the (*u*,*v*) texture coordinates for the vertex index; the `tex_coord.Z_` element is not the third texture coordinate *w*, which we don't need, but is instead the camera space *z* value, which we do need.

The overridden methods `clip_segment_to_near_z` and `clip_near_z` perform the same functions as in the parent class `l3d_polygon_3d`, but have been extended also to clip the *u* and *v* coordinates against the near *z* plane. Near *z* clipping has the effect of finding edges which cross the near *z* plane, and clipping these edges to become shorter so that they do not recede beyond the near *z* plane. This means that the 3D location of the endpoints of the clipped edges changes. Since we have now associated texture coordinates with the endpoints of these edges, clipping off the endpoints and replacing them with new endpoints means that we also must clip the texture coordinates and assign them to the new clipped endpoint. This is very simple; during normal clipping of the geometry, we already compute a time parameter `t` indicating the "time" of the intersection, which is a relative value indicating the percentage displacement between the starting point and the ending point. We multiply the time parameter with the *x*, *y*, and *z* displacements to find the exact location of the clipped point. To clip the *u* and *v* values, we also use the same time parameter, and multiply it with the *u* displacement and *v* displacement to obtain the (*u*,*v*) coordinates at the new clipped point.

Similarly, the overridden methods `clip_segment_to_edge_2d` and `clip_to_edge_2d` have been extended not only to clip the polygon in 2D, but also to clip the texture coordinates in 2D. The only tricky part is that we cannot clip the *u* and *v* values directly in 2D; instead, we must compute the u/z, v/z, and $1/z$ values, clip these by using the same time parameter `t` used to clip the geometry, then divide u/z and v/z by $1/z$ to recover the clipped *u* and *v* values. This is different than the near *z* clipping case; near *z* clipping occurs in 3D, where the *u* and *v* values have a

linear relationship with the 3D x, y, and z values, as evidenced by the linear equations in the world to texture space matrix. But for 2D clipping, the clipping occurs in 2D screen coordinates; the u and v values do not have a linear relationship to the screen coordinates, but the u/z, v/z, and $1/z$ values are linear in screen space. This is why we clip these values, and not the original u and v values, in 2D.

The method `clip_to_plane`, which clips the polygon against an arbitrary plane in 3D (contrast with the 2D clipping of the previous paragraph), is also overridden to clip the texture coordinates in 3D in addition to the geometry. Since there is ordinarily a linear relationship between texture space and 3D world space (unless we manually assign texture coordinates; see Chapter 3), we directly clip the u and v values in 3D, in contrast to the u/z, v/z, and $1/z$ clipping necessary in 2D. As we saw earlier, after clipping the crossing polygon segments in 3D against the clipping plane, the clipping plane stores an `intersection_t` parameter representing the percentage displacement of the new clipped point along the path from the start point in the crossing segment to the end point. To clip the u and v values in 3D, we simply use this same `intersection_t` parameter to compute new u and v values as the same percentage displacement along the path from the starting (u,v) values at the first vertex in the crossing segment to the ending (u,v) values at the end vertex in the crossing segment. In other words, to perform clipping in 3D, we simply linearly interpolate the (u,v) values just as we do with the world coordinate locations.

The overridden method `draw` causes the polygon to draw itself; the polygon, in turn, asks its rasterizer to draw the polygon. This means that all existing code which uses polygons—such as the entire 3D object class—can handle texture mapped polygons with no change; the virtual call to the `draw` routine always gets bound at run time to the correct polygon-specific drawing routine.

Now, let's look at the rasterization classes that actually draw the polygons on-screen.

Rasterization Classes for 3D Polygons

We've now seen everything we need to store the data for texture mapping. It's now time to see the actual texture mapping routines themselves. These are located within the rasterizer classes.

An Abstract 3D Rasterizer: l3d_rasterizer_3d

Class `l3d_rasterizer_3d` is a 3D rasterizer class that forwards requests for rasterization on to a 3D rasterizer implementation. (Recall that the bridge design pattern separates the rasterizer and its implementation.)

Listing 2-8: `rast3.h`

```
#ifndef __RAST3_H
#define __RAST3_H
#include "../tool_os/memman.h"

#include "rasteriz.h"
#include "../tool_2d/scrinfo.h"

#include "../geom/polygon/p3_flat.h"
#include "../geom/texture/texture.h"

class l3d_polygon_3d_textured;
class l3d_polygon_3d_textured_lightmapped;
```

```
 class l3d_rasterizer_3d_imp :
      virtual public l3d_rasterizer_2d_imp,
      virtual public l3d_texture_computer
{
  public:
    l3d_rasterizer_3d_imp(int xs, int ys, l3d_screen_info *si);

    /* virtual */ void clear_buffer(void) {};

    virtual void draw_polygon_textured(const l3d_polygon_3d_textured *p_poly);
    virtual void draw_polygon_textured_lightmapped
    (const l3d_polygon_3d_textured_lightmapped *p_poly);
    virtual void draw_text(int x, int y, const char *string) {};
    virtual void set_text_color(int red, int green, int blue) {};
};

class l3d_rasterizer_3d : public l3d_rasterizer_2d {
  protected:
    l3d_rasterizer_3d_imp *imp3d;

  public:
    l3d_rasterizer_3d(l3d_rasterizer_2d_imp *i2,
                      l3d_rasterizer_3d_imp *i3) :
    l3d_rasterizer_2d(i2)
    {
      imp3d = i3;
    }
    virtual ~l3d_rasterizer_3d(void) {}

    void draw_point( int x , int y ,  unsigned long col ) {
      imp3d->draw_point(x,y,col);
    }

    /* virtual */ void draw_line( int x0 , int y0 , int x1 ,  int y1 , unsigned long col ) {
      imp3d->draw_line(x0,y0,x1,y1,col);
    }

    /* virtual */ void clear_buffer(void) {
      imp3d->clear_buffer();
    }

    /* virtual */ void draw_polygon_flatshaded( const l3d_polygon_3d_flatshaded
        *p_poly )
  {imp3d->draw_polygon_flatshaded(p_poly); }

    virtual void draw_polygon_textured(const l3d_polygon_3d_textured *p_poly)
  {imp3d->draw_polygon_textured(p_poly); }

    virtual void draw_polygon_textured_lightmapped
    (const l3d_polygon_3d_textured_lightmapped *p_poly)
  {imp3d->draw_polygon_textured_lightmapped(p_poly); }

    void draw_text(int x, int y, const char *string)
  {imp3d->draw_text(x, y, string); }

};

class l3d_rasterizer_3d_imp_factory {
  public:
    virtual l3d_rasterizer_3d_imp *create(int xs, int ys, l3d_screen_info *si)=0;
```

```
};

#ifndef __ACTIVE__P3_TEX_H
#ifndef __ACTIVE__P3_CLIP_H
#include "../geom/polygon/p3_tex.h"
#include "../geom/polygon/p3_ltex.h"
#endif
#endif

#endif
```

Listing 2-9: `rast3.cc`

```
#include "rast3.h"
#include "../tool_os/memman.h"

l3d_rasterizer_3d_imp::l3d_rasterizer_3d_imp
(int xs, int ys, l3d_screen_info *si) :
    l3d_rasterizer_2d_imp(xs,ys,si)
{}

void l3d_rasterizer_3d_imp::draw_polygon_textured
(const l3d_polygon_3d_textured *p_poly)
{}

void l3d_rasterizer_3d_imp::draw_polygon_textured_lightmapped
(const l3d_polygon_3d_textured_lightmapped *p_poly)
{}
```

The member variable `imp3d` is a pointer to a 3D rasterizer implementation object, which actually carries out the rasterizer's drawing requests. The 3D rasterizer implementation object is of (abstract) type `l3d_rasterizer_3d_imp`, and offers the new methods `draw_polygon_textured` and `draw_polygon_textured_lightmapped`. These methods are empty in the abstract `l3d_rasterizer_3d_imp` class; they are overridden in the actual concrete 3D rasterizer implementation (see next section).



NOTE The factory manager class `l3d_factory_manager_v_0_2`, which we saw earlier in this chapter, creates and manages an object of type `l3d_rasterizer_3d_imp`. Thus, it is through the factory manager that the application chooses which 3D rasterizer implementation (software or OpenGL/Mesa) to use.

The overridden methods `draw_point`, `draw_line`, `clear_buffer`, and `draw_polygon_flatshaded` perform the same functions as in the parent class `l3d_rasterizer_2d`, only the requests are forwarded to the 3D rasterizer implementation (`imp3d`) instead of to the 2D rasterizer (`imp2d`).

The new methods `draw_polygon_textured` and `draw_polygon_textured_lightmapped` are the new functionality offered by the 3D rasterizer interface. These methods draw a textured and a light mapped polygon, respectively, by forwarding the request on to the rasterizer implementation.

The new methods `set_text_color` and `draw_text` are used for drawing text strings into the rasterizer's buffer for display. We haven't needed to until now, but it can be useful to display diagnostic or other messages to the screen; since the screen contents are controlled by the rasterizer, we need an interface in the rasterizer (and in the rasterizer implementation) offering text

drawing capabilities. This is exactly what these virtual text functions do; they are overridden in the concrete rasterizer implementation subclasses covered below.

A Software 3D Rasterizer Implementation: l3d_rasterizer_3d_sw_imp

The class `l3d_rasterizer_3d_sw_imp` is a software implementation of the rasterization services required by class `l3d_rasterizer_3d`.

The main function of interest is the new function `draw_polygon_textured`. Fundamentally, it draws a polygon in much the same manner as we draw flat-shaded polygons: we proceed from top to bottom, rasterizing each edge on the left and right sides of the polygon simultaneously. Between the left edge and the right edge we draw a horizontal span of pixels. Figure 2-39 summarizes the idea behind rasterizing flat-shaded polygons. The current left edge and current right edge are drawn in bold; the left edge goes horizontally from `left_x_start` to `left_x_end`; the right edge, from `right_x_start` to `right_x_end`. The current scanline is the horizontal line labeled with `y`. For the current scanline, we have the current value of the left edge `left_x`, the current value of the right edge `right_x`, and the current horizontal position `x`. Within the scanline we move from `left_x` to `right_x` by increments of one pixel. After finishing drawing the current scanline, we move down to the next scanline, labeled with `y+1`; at this point, we also move the left edge horizontally by `left_dx_dy` and the right edge by `right_dx_dy`. After finishing drawing the left edge and the right edge, the left edge reaches the horizontal value `left_x_end`; the right edge reaches `right_x_end`.

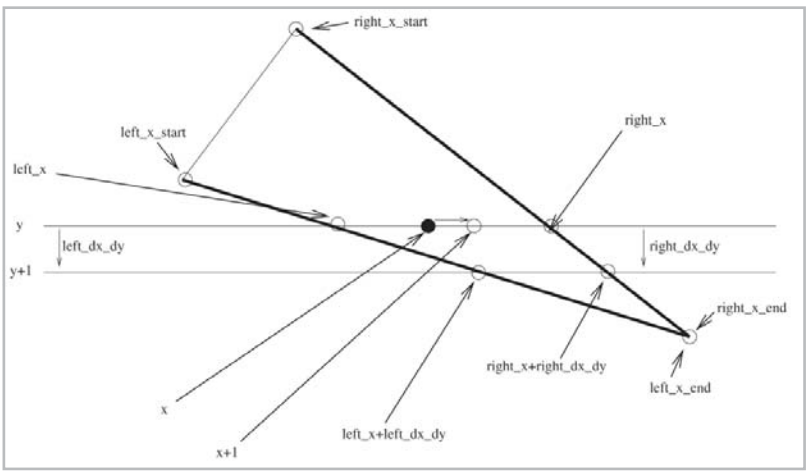


Figure 2-39: Variables for rasterizing flat-shaded polygons.

The main difference between the previous flat-shaded rasterization code and the current texture mapping code is that now, while stepping along the edges vertically and the span horizontally, we must also keep track of the current u/z, v/z, and $1/z$ values. We do this using two sets of variables, as mentioned earlier: one for the left edge of the polygon and one for the right edge of the polygon. Recall, the polygon drawing code works by searching simultaneously for edges of non-zero height on the left and right sides of the polygon, and by drawing horizontal spans of

pixels from top to bottom between the left and right edges. With texture mapping, as soon as we find an edge to be drawn for the left or the right side, we immediately make note of the u/z, v/z, and $1/z$ values for both the starting vertex and the ending vertex of the edge. Then, we compute three delta values indicating how much the u/z, v/z, and $1/z$ values change from one scanline to the next by dividing the total change by the height of the edge. In this manner, we always know the u/z, v/z, and $1/z$ values at the left and right sides of the current scanline. Then, within the scanline, we draw one horizontal span of pixels from the left edge to the right edge. Based on the left and right u/z, v/z, and $1/z$ values, and the length of the horizontal span in pixels, we can interpolate the values from the left edge to the right edge. This gives us a u/z, v/z, and $1/z$ value for each pixel. We then simply divide u/z by $1/z$ and v/z by $1/z$ to obtain the u and v values, scale these real-valued u and v values to integer texture map coordinates, retrieve the pixel from the texture map, and draw the pixel on-screen. We actually divide the pixels into horizontal runs of 16 pixels, only computing the u and v values every 16th pixel and linearly interpolating the u and v values between the ends of the run.

See Figure 2-40 for a summary of the new variables needed while rasterizing texture mapped polygons. Notice that this is simply an extension of the flat-shaded case; we keep track of additional variables at the starting and ending vertices of the left and right edges, interpolate these values vertically between scanlines to obtain values for the left and right edges of the scanline, and also interpolate these values horizontally within the scanline between the left and right values of the current scanline.

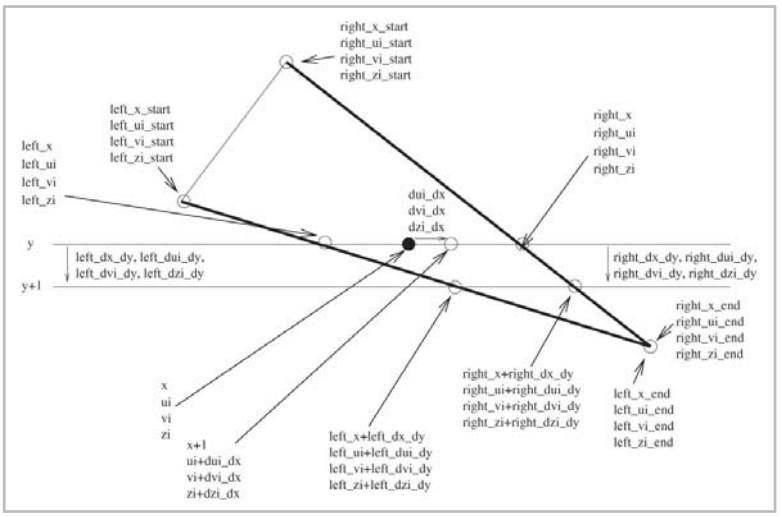


Figure 2-40: Variables for rasterizing texture mapped polygons.

Listing 2-10: ras3_sw.h

```
#ifndef __RAS3_SW_H
#define __RAS3_SW_H
#include "../tool_os/memman.h"

#include "ras_sw.h"
#include "rast3.h"
#include "math.h"
```

```
#include "../system/sys_dep.h"

class l3d_rasterizer_3d_sw_imp :
      virtual public l3d_rasterizer_2d_sw_imp,
      virtual public l3d_rasterizer_3d_imp
{
  protected:
    void compute_surface_orientation_and_size(void);
    unsigned char *text_color;
  public:
    l3d_rasterizer_3d_sw_imp(int xs, int ys, l3d_screen_info *si);
    virtual ~l3d_rasterizer_3d_sw_imp(void);

    /* virtual */ void draw_polygon_textured(const l3d_polygon_3d_textured *p_poly);
    /* virtual */ void draw_polygon_textured_lightmapped
    (const l3d_polygon_3d_textured_lightmapped *p_poly);
    /* virtual */ void draw_text(int x, int y, const char *text);
    /* virtual */ void set_text_color(int red, int green, int blue);
    /* virtual */ void clear_buffer(void);

};

class l3d_rasterizer_3d_sw_imp_factory :
      public l3d_rasterizer_3d_imp_factory
{
  public:
    l3d_rasterizer_3d_imp *create(int xs, int ys, l3d_screen_info *si);
};

#endif
```

Listing 2-11: ras3_sw.cc

```
#include "ras3_sw.h"
#include "../system/sys_dep.h"
#include <string.h>
#include <stdlib.h>
#include "../tool_os/memman.h"

static const unsigned char period[] =
  {0x00, 0x00, 0x18, 0x18, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

static const unsigned char space[] =
  {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

static const unsigned char digits[][13] = {
  {0x0,0x0, 0x18,0x3C,0x66,0xC3,0xC3,0xC3,0xC3,0xC3,0x66,0x3C,0x18},
  {0x0,0x0, 0x7F,0x7F,0xC,0xC,0xC,0xC,0xC,0xC,0x6C,0x3C,0xC},
  {0x0,0x0, 0xFF,0xFF,0xC0,0x60,0x30,0x18,0xC,0x6,0xC6,0x66,0x3C},
  {0x0,0x0, 0x7E,0xC3,0x3,0x6,0xC,0x38,0xC,0x6,0x3,0xC3,0x7E},
  {0x0,0x0, 0x6,0x6,0x6,0x6,0xFF,0xFF,0xC6,0xC6,0x66,0x36,0x1E},
  {0x0,0x0, 0x7E,0xFF,0xC3,0x3,0x3,0xFF,0xFE,0xC0,0xC0,0xC0,0xFE},
  {0x0,0x0, 0x7E,0xFF,0xC3,0xC3,0xC3,0xFF,0xFE,0xC0,0xC0,0xC0,0x7E},
  {0x0,0x0, 0x60,0x60,0x60,0x60,0x30,0x18,0xC,0x6,0x3,0x3,0xFF},
  {0x0,0x0, 0x7E,0xFF,0xC3,0xC3,0xC3,0x7E,0xC3,0xC3,0xC3,0xC3,0x7E},
  {0x0,0x0, 0x7E,0xFF,0xC3,0x3,0x3,0x7F,0xC3,0xC3,0xC3,0xC3,0x7E}
};

static const unsigned char letters[][13] = {
  {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0x66, 0x3c, 0x18},
  {0x00, 0x00, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
```

```
  {0x00, 0x00, 0x7e, 0xe7, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
  {0x00, 0x00, 0xfc, 0xce, 0xc7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc7, 0xce, 0xfc},
  {0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xc0, 0xff},
  {0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xff},
  {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xcf, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
  {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
  {0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x7e},
  {0x00, 0x00, 0x7c, 0xee, 0xc6, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06},
  {0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xe0, 0xf0, 0xd8, 0xcc, 0xc6, 0xc3},
  {0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0},
  {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xdb, 0xff, 0xff, 0xe7, 0xc3},
  {0x00, 0x00, 0xc7, 0xc7, 0xcf, 0xcf, 0xdf, 0xdb, 0xfb, 0xf3, 0xf3, 0xe3, 0xe3},
  {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xe7, 0x7e},
  {0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
  {0x00, 0x00, 0x3f, 0x6e, 0xdf, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0x66, 0x3c},
  {0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
  {0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0xe0, 0xc0, 0xc0, 0xe7, 0x7e},
  {0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0xff},
  {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
  {0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
  {0x00, 0x00, 0xc3, 0xe7, 0xff, 0xff, 0xdb, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
  {0x00, 0x00, 0xc3, 0x66, 0x66, 0x3c, 0x3c, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3},
  {0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3},
  {0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x7e, 0x0c, 0x06, 0x03, 0x03, 0xff}
};

unsigned char font_bitmaps[256][13];

l3d_rasterizer_3d_imp * l3d_rasterizer_3d_sw_imp_factory::create
(int xs, int ys, l3d_screen_info *si)
{
  return new l3d_rasterizer_3d_sw_imp(xs,ys,si);
}

l3d_rasterizer_3d_sw_imp::l3d_rasterizer_3d_sw_imp
(int xs, int ys, l3d_screen_info *si)
    : l3d_rasterizer_3d_imp(xs,ys,si),
    l3d_rasterizer_2d_sw_imp(xs,ys,si),
    l3d_rasterizer_2d_imp(xs,ys,si)

{

  text_color = new unsigned char [si->bytes_per_pixel];
  set_text_color(si->ext_max_red, si->ext_max_green, si->ext_max_blue);

  //- create font bitmaps
  int i,j;

  for(i=0; i<26; i++) {
    for(j=0; j<13; j++) {
      font_bitmaps['A' + i][j] = letters[i][j];
      font_bitmaps['a' + i][j] = letters[i][j];
    }
  }

  for(i=0; i<10; i++) {
    for(j=0; j<13; j++) {
      font_bitmaps['0' + i][j] = digits[i][j];
    }
  }
```

Team-Fly®

```
   for(j=0; j<13; j++) {
    font_bitmaps[' '][j] = space[j];
  }

  for(j=0; j<13; j++) {
    font_bitmaps['.'][j] = period[j];
  }

}

l3d_rasterizer_3d_sw_imp::~l3d_rasterizer_3d_sw_imp(void) {
  delete [] text_color;
}

void l3d_rasterizer_3d_sw_imp::clear_buffer(void)
{
  l3d_rasterizer_2d_sw_imp::clear_buffer();
}

#define Z_MULT_FACTOR 512
#define Z_ADD_FACTOR -0.42

//- convenience macro for accessing the vertex coordinates. Notice
//- that we access the clipped vertex index list (not the original),
//- and the transformed coordinate (not the original). This means
//- we draw the clipped version of the transformed polygon.
#define VTX(i) ((**(p_poly->vlist))[ (*(p_poly->clip_ivertices))[i].ivertex ].transformed)

#include "../math/fix_lowp.h"
#if 0
#define l3d_fixed float
#define l3d_real_to_fixed(x) (x)
#define fixfixdiv(a,b) ((a)/(b))
#define fixfixmul(a,b) ((a)*(b))
#define int2fix(a) ( (float)(a))
#define fix2int(a) ( (int)(a))
#define fix2float(a) (a)
#define float2fix(a) (a)
#define iceil_fix(a) ( (int)ceil((double)(a)) )

#endif

void l3d_rasterizer_3d_sw_imp::draw_polygon_textured
(const l3d_polygon_3d_textured *p_poly)
{

  l3d_fixed x0,y0,x1,y1,x2,y2,x3;
  l3d_fixed top_y, bottom_y;
  int point_on_right=0;
  int left_idx, right_idx, top_y_idx, bottom_y_idx;

  int maxy_upper, iceil_fix_y0, iceil_fix_y1, iceil_fix_y2;

  l3d_point *vtemp;

  int i;

  int scanline;

  //- variables for the left edge, incremented BETWEEN scanlines
```

```
//- (inter-scanline)

l3d_fixed volatile
left_x,                     left_ui,       left_vi,       left_zi,
left_x_start, left_y_start, left_ui_start, left_vi_start, left_zi_start,
left_x_end,   left_y_end,   left_ui_end,   left_vi_end,   left_zi_end,
left_dx_dy,                 left_dui_dy,   left_dvi_dy,   left_dzi_dy;
int
left_ceilx,
left_ceilx_start, left_ceilx_end,
left_ceily_start, left_ceily_end;

//- variables for the right edge, incremented BETWEEN scanlines
//- (inter-scanline)

l3d_fixed volatile
right_x,                    right_ui,      right_vi,      right_zi,
right_x_start,right_y_start,right_ui_start,right_vi_start,right_zi_start,
right_x_end,  right_y_end,  right_ui_end,  right_vi_end,  right_zi_end,
right_dx_dy,                right_dui_dy,  right_dvi_dy,  right_dzi_dy;

int
right_ceilx,
right_ceilx_start, right_ceilx_end,
right_ceily_start, right_ceily_end;

//- variables incremented WITHIN one scanline (intra-scanline)

l3d_fixed volatile
u,v,z,
ui, vi, zi, dui_dx, dvi_dx, dzi_dx,

//- for linear interp. within one run
u_left, u_right, du_dx,
v_left, v_right, dv_dx, inv_dx,
denom, inv_run_dx,

run_x, run_x_start, run_x_end,
u_run_end,  v_run_end, du_dx_run, dv_dx_run;

long int cur_x;

top_y = l3d_real_to_fixed(VTX(0).Y_);
top_y_idx = 0;
bottom_y = top_y;
bottom_y_idx = top_y_idx;
for(i=0; i<p_poly->clip_ivertices->num_items; i++) {
  if(l3d_real_to_fixed(VTX(i).Y_) < top_y) {
    top_y = l3d_real_to_fixed(VTX(i).Y_);
    top_y_idx = i;
  }
  if(l3d_real_to_fixed(VTX(i).Y_) > bottom_y) {
    bottom_y = l3d_real_to_fixed(VTX(i).Y_);
    bottom_y_idx = i;
  }
}

left_idx = top_y_idx;
right_idx = top_y_idx;
```

```
  left_x_start=l3d_real_to_fixed(VTX(top_y_idx).X_);
  left_ceilx_start=iceil_fix(left_x_start);
  left_y_start=l3d_real_to_fixed(VTX(top_y_idx).Y_);
  left_ceily_start=iceil_fix(left_y_start);
  left_ceily_end=left_ceily_start;
  left_zi_start = fixfixdiv(int2fix(Z_MULT_FACTOR),
                    l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                       & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Z_)
                          + float2fix(Z_ADD_FACTOR));
  left_zi_end = left_zi_start;
  left_ui_start = fixfixmul(
                    l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                       & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.X_),
      left_zi_start);
  left_ui_end = left_ui_start;
  left_vi_start = fixfixmul(
                    l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                       & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Y_),
                    left_zi_start);
  left_vi_end = left_vi_start;

  right_x_start=left_x_start;
  right_y_start=left_y_start;
  right_ceily_start=left_ceily_start;
  right_ceily_end=right_ceily_start;
  right_ui_start = left_ui_start;
  right_vi_start = left_vi_start;

  right_zi_start = left_zi_start;

  scanline = left_ceily_start;
  int oneshot=1;

  while(scanline < ysize) {
    //- if needed, find next left-edge whose ceily_end > current scanline
    while( left_ceily_end - scanline <= 0 )
    {
      if (left_idx == bottom_y_idx) {
        return;
      }
      left_idx = p_poly->next_clipidx_left(left_idx,p_poly->clip_ivertices->num_items);
      left_y_end=l3d_real_to_fixed(VTX(left_idx).Y_);
      left_ceily_end = iceil_fix(left_y_end);
      if(left_ceily_end - scanline) {
#define MIN_EDGE_HEIGHT float2fix(0.005)
#define MIN_EDGE_WIDTH float2fix(0.005)
        left_x_end=l3d_real_to_fixed(VTX(left_idx).X_);
        left_ceilx_end=iceil_fix(left_x_end);

        left_zi_end = fixfixdiv(int2fix(Z_MULT_FACTOR),
                        l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                           & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Z_)
                              + float2fix(Z_ADD_FACTOR));
        left_ui_end = fixfixmul(
                        l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                           & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.X_),
                        left_zi_end);
        left_vi_end = fixfixmul(
                        l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                           & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Y_),
```

```
                    left_zi_end);

    if(left_y_end - left_y_start >= MIN_EDGE_HEIGHT) {
      left_dx_dy = fixfixdiv(left_x_end-left_x_start,left_y_end-left_y_start);

      left_dui_dy = fixfixdiv(left_ui_end - left_ui_start,
                              left_y_end  - left_y_start);
      left_dvi_dy = fixfixdiv(left_vi_end - left_vi_start,
                              left_y_end  - left_y_start);
      left_dzi_dy = fixfixdiv(left_zi_end - left_zi_start,
                              left_y_end  - left_y_start);

    }
    else
    {
      left_dx_dy =
        left_dui_dy =
          left_dvi_dy =
            left_dzi_dy = int2fix(0);
    }

    left_x = left_x_start +   //- sub-pixel correction
             fixfixmul(int2fix(left_ceily_start)-left_y_start , left_dx_dy);
    left_ui = left_ui_start + //- sub-pixel correction
              fixfixmul(int2fix(left_ceily_start)-left_y_start,
                        left_dui_dy);
    left_vi = left_vi_start + //- sub-pixel correction
              fixfixmul(int2fix(left_ceily_start)-left_y_start,
                        left_dvi_dy);
    left_zi = left_zi_start + //- sub-pixel correction
              fixfixmul(int2fix(left_ceily_start)-left_y_start,
                        left_dzi_dy);
  }else {
    left_x_start = l3d_real_to_fixed(VTX(left_idx).X_);
    left_ceilx_start=iceil_fix(left_x_start);
    left_y_start = l3d_real_to_fixed(VTX(left_idx).Y_);
    left_ceily_start = iceil_fix(left_y_start);
    left_zi_start = fixfixdiv(int2fix(Z_MULT_FACTOR),
                      l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                         & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Z_)
                             + float2fix(Z_ADD_FACTOR));
    left_ui_start = fixfixmul(
                      l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                         & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.X_),
                      left_zi_start);
    left_vi_start = fixfixmul(
                      l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                         & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Y_),
                      left_zi_start);

  }
}

//- if needed, find next right-edge whose ceily_end > current scanline
while(right_ceily_end - scanline <= 0 )
{
  if (right_idx == bottom_y_idx) {
    return;
  }
  right_idx = p_poly->next_clipidx_right(right_idx, p_poly->clip_ivertices->num_items);
```

```
right_y_end=l3d_real_to_fixed(VTX(right_idx).Y_);
right_ceily_end = iceil_fix(right_y_end);
if(right_ceily_end - scanline) {

  right_x_end=l3d_real_to_fixed(VTX(right_idx).X_);
  right_ceilx_end = iceil_fix(right_x_end);
  right_zi_end = fixfixdiv(int2fix(Z_MULT_FACTOR),
                  l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                     & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.Z_)
                         + float2fix(Z_ADD_FACTOR));
  right_ui_end = fixfixmul(
                  l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                     & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.X_),
                  right_zi_end);
  right_vi_end = fixfixmul(
                  l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                     & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.Y_),
                  right_zi_end);

  if(right_y_end-right_y_start>=MIN_EDGE_HEIGHT) {
    right_dx_dy =
      fixfixdiv(right_x_end-right_x_start,right_y_end-right_y_start);

    right_dui_dy = fixfixdiv(right_ui_end - right_ui_start,
                             right_y_end  - right_y_start);
    right_dvi_dy = fixfixdiv(right_vi_end - right_vi_start,
                             right_y_end  - right_y_start);
    right_dzi_dy = fixfixdiv(right_zi_end - right_zi_start,
                             right_y_end  - right_y_start);

  }
  else
  {
    right_dx_dy =
      right_dui_dy =
        right_dvi_dy =
          right_dzi_dy = int2fix(0);
  }

  right_x = right_x_start +  //- sub-pixel correction
            fixfixmul(int2fix(right_ceily_start)-right_y_start , right_dx_dy);

  right_ui = right_ui_start + //- sub-pixel correction
             fixfixmul(int2fix(right_ceily_start)-right_y_start,
                       right_dui_dy);
  right_vi = right_vi_start + //- sub-pixel correction
             fixfixmul(int2fix(right_ceily_start)-right_y_start,
                       right_dvi_dy);
  right_zi = right_zi_start + //- sub-pixel correction
             fixfixmul(int2fix(right_ceily_start)-right_y_start,
                       right_dzi_dy);

}else {
  right_x_start = l3d_real_to_fixed(VTX(right_idx).X_);
  right_ceilx_start = iceil_fix(right_x_start);
  right_y_start = l3d_real_to_fixed(VTX(right_idx).Y_);
  right_ceily_start = iceil_fix(right_y_start);

  right_zi_start = fixfixdiv(int2fix(Z_MULT_FACTOR),
                    l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
```

```
                                        & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.Z_)
                                  +float2fix(Z_ADD_FACTOR));
        right_ui_start = fixfixmul(
                          l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                        & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.X_),
                          right_zi_start);
        right_vi_start = fixfixmul(
                          l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                        & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.Y_),
                          right_zi_start);
      }
    }

    if (left_ceily_end > ysize) left_ceily_end = ysize;
    if (right_ceily_end > ysize) right_ceily_end = ysize;

    while ( (scanline < left_ceily_end) && (scanline < right_ceily_end) )
    {
      if (left_x > right_x) {
      }
      else {
        left_ceilx = iceil_fix(left_x);
        right_ceilx = iceil_fix(right_x);

        //- if the segment is (horizontally) visible, draw it.

        if(right_x - left_x > MIN_EDGE_WIDTH) {

          dui_dx = fixfixdiv(right_ui - left_ui , right_x  - left_x);
          dvi_dx = fixfixdiv(right_vi - left_vi , right_x  - left_x);
          dzi_dx = fixfixdiv(right_zi - left_zi , right_x  - left_x);

        }else {
          //- a segment exactly one pixel wide (left_x == right_x)
          dui_dx =
            dvi_dx =
              dzi_dx = int2fix(0);
        }

        ui = left_ui + //- sub-pixel correction
             fixfixmul(int2fix(left_ceilx) - left_x, dui_dx);
        vi = left_vi + //- sub-pixel correction
             fixfixmul(int2fix(left_ceilx) - left_x, dvi_dx);
        zi = left_zi + //- sub-pixel correction
             fixfixmul(int2fix(left_ceilx) - left_x, dzi_dx);

        cur_x=left_ceilx;

#define LINEAR_RUN_SHIFT 4
#define LINEAR_RUN_SIZE (1<<LINEAR_RUN_SHIFT)
        z = fixfixdiv(int2fix(1), zi);
        u = fixfixmul(ui,z);
        v = fixfixmul(vi,z);

        for(/*register */ unsigned char *pix=sinfo->p_screenbuf+(left_ceilx + SW_RAST_Y_REVERSAL(ysize,
                                          scanline)*xsize)*sinfo->bytes_per_pixel;
                          pix< sinfo->p_screenbuf+(right_ceilx+ SW_RAST_Y_REVERSAL(ysize,
                                             scanline)*xsize)*sinfo->bytes_per_pixel;
           )
        {
```

```
          run_x_end = int2fix(cur_x) +
                      int2fix(LINEAR_RUN_SIZE);
          if (run_x_end > int2fix(right_ceilx)) {
            run_x_end = int2fix(right_ceilx);
          }
          denom = fixfixdiv
                  (int2fix(1) ,
                   zi + fixfixmul(dzi_dx, run_x_end-int2fix(cur_x)));
          u_run_end=fixfixmul
                    (ui + fixfixmul(dui_dx,(run_x_end-int2fix(cur_x))),
                     denom) ;
          v_run_end=fixfixmul
                    (vi + fixfixmul(dvi_dx,(run_x_end-int2fix(cur_x))),
                     denom) ;

          inv_run_dx = fixfixdiv
                       (int2fix(1),
                        (run_x_end-int2fix(cur_x)));
          du_dx_run = fixfixmul((u_run_end - u) ,inv_run_dx);
          dv_dx_run = fixfixmul((v_run_end - v) ,inv_run_dx);

          for(run_x = int2fix(cur_x);
              run_x < run_x_end;
              run_x+=int2fix(1))
          {

            unsigned char *texel =
              p_poly->texture->tex_data->data +
              ((fix2int(l3d_mulri(v,p_poly->texture->tex_data->height-1))&
              (p_poly->texture->tex_data->height-1))*(p_poly->texture->tex_data->width)
               +
              (fix2int(l3d_mulri(u,p_poly->texture->tex_data->width-1))&
              (p_poly->texture->tex_data->width-1)) ) *
              sinfo->bytes_per_pixel;

            for(register int b=0; b<sinfo->bytes_per_pixel;b++) {
              *pix++ =  *texel++;
            }
            u += du_dx_run;
            v += dv_dx_run;
          }

          cur_x += LINEAR_RUN_SIZE;
#if 0
          ui += dui_dx<<LINEAR_RUN_SHIFT;
          vi += dvi_dx<<LINEAR_RUN_SHIFT;
          zi += dzi_dx<<LINEAR_RUN_SHIFT;
#else
ui += l3d_mulri(dui_dx, LINEAR_RUN_SIZE);
          vi += l3d_mulri(dvi_dx, LINEAR_RUN_SIZE);
          zi += l3d_mulri(dzi_dx, LINEAR_RUN_SIZE);
#endif
        }
      }

      scanline++;
      left_x += left_dx_dy;
      right_x += right_dx_dy;

      left_ui += left_dui_dy;
```

```
      left_vi += left_dvi_dy;
      left_zi += left_dzi_dy;
      right_ui += right_dui_dy;
      right_vi += right_dvi_dy;
      right_zi += right_dzi_dy;
    }

    //- for the left and/or right segment(s) which just completed drawing
    //- initialize xxx_start = xxx_end, to begin next segment. xxx_end is
    //- then searched for in the next iteration (the while() loops)
    if ( left_ceily_end - scanline <= 0 ) {
      left_x_start=left_x_end;
      left_y_start=left_y_end;
      left_ceily_start=left_ceily_end;
      left_ui_start=left_ui_end;
      left_vi_start=left_vi_end;
      left_zi_start=left_zi_end;

    }
    if ( right_ceily_end - scanline <= 0 ) {
      right_x_start=right_x_end;
      right_y_start=right_y_end;
      right_ceily_start=right_ceily_end;
      right_ui_start=right_ui_end;
      right_vi_start=right_vi_end;
      right_zi_start=right_zi_end;

    }
  }

}
#include "../math/fix_prec.h"

void l3d_rasterizer_3d_sw_imp::draw_polygon_textured_lightmapped
(const l3d_polygon_3d_textured_lightmapped *p_poly)
{
  l3d_texture *old_tex;

  if(p_poly->surface==NULL) {
    p_poly->surface = p_poly->get_scache()->combine_lightmap_and_texmap
                      (p_poly->surface_xsize, p_poly->surface_ysize, p_poly);
  }

  p_poly->surface->O = p_poly->surface_orientation.O;
  p_poly->surface->U = p_poly->surface_orientation.U;
  p_poly->surface->V = p_poly->surface_orientation.V;

  old_tex = p_poly->texture;
  p_poly->texture = p_poly->surface;
  draw_polygon_textured(p_poly);
  p_poly->texture = old_tex;
}

void l3d_rasterizer_3d_sw_imp::set_text_color(int red, int green, int blue) {

  unsigned long native_col = sinfo->ext_to_native
                             (red, green, blue);

  register int i;
  unsigned long mask = MAX_BYTE;
```

```
  char shift = 0;

  unsigned char *c = text_color;
  for(i=0; i<sinfo->bytes_per_pixel; i++) {
    *c++ = (native_col & mask) >> shift;
    mask <<= BITS_PER_BYTE;
    shift += BITS_PER_BYTE;
  }
}

void l3d_rasterizer_3d_sw_imp::draw_text(int x, int y, const char *text) {

#define FONT_Y_SIZE 13
#define FONT_X_SIZE 8
#define FONT_SPACING 2

  int i;
  int font_y, font_x;
  int x_offset;

  x_offset = x * sinfo->bytes_per_pixel;
  unsigned char *pix;

  unsigned char *fg_color;

  for(font_y=0; font_y < FONT_Y_SIZE; font_y++) {
    pix=sinfo->p_screenbuf+(x_offset +
                            SW_RAST_Y_REVERSAL(ysize,y + font_y)*xsize)
        *sinfo->bytes_per_pixel;

    unsigned int c;
    for(c=0; c<strlen(text); c++) {
      int current_bit;
      for(font_x=0, current_bit=0x80;
          font_x < FONT_X_SIZE;
          font_x++, current_bit >>= 1)
      {
        if(font_bitmaps[text[c]][FONT_Y_SIZE-font_y] & current_bit) {
          fg_color = text_color;
          for(register int b=0; b<sinfo->bytes_per_pixel;b++) {
            *pix++ =  *fg_color++;
          }
        }else {
          pix += sinfo->bytes_per_pixel;
        }
      }
      for(font_x=0; font_x < FONT_SPACING; font_x++) {
        pix += sinfo->bytes_per_pixel;
      }
    }
  }
}
```

Two other items deserve special mention in the texture mapping code. First, all of the real-valued variables are of type `l3d_fixed` instead of `l3d_real`. This means that we explicitly force the usage of fixed-point values within the texture mapping code. As with `l3d_real` values, we also use macros to multiply and divide `l3d_fixed` values: `fixfixmul` and `fixintmul` multiply a fixed point by another fixed-point or integer value, and `fixfixdiv` and `fixintdiv` divide a fixed-point value by another fixed-point or integer value. Addition and subtraction of `l3d_`

`fixed` values, as with `l3d_real`, can be done with the normal C++ addition and subtraction operators.

The reason we use fixed-point values in the texture mapping code is speed. Fixed-point arithmetic represents real-valued numbers as integers (see the Appendix); therefore, the conversion from a fixed-point number to an integer is quite fast—it is simply a bit shift operation. On the other hand, converting a floating-point number to an integer is comparatively slow. This might seem trivial, but we must convert the real-valued (u,v) coordinates to integer texture map indices for every pixel. In the l3d library, for instance, simply changing from floating point to fixed point doubled the performance of the texture mapping code.

The other item of note is the use of the `Z_MULT_FACTOR` and `Z_ADD_FACTOR` constants in the code. Instead of computing $1/z$, we compute `Z_MULT_FACTOR`$/(z +$ `Z_ADD_FACTOR`$)$. The $1/z$ values in camera space can be very small; for instance, if z is 1000, then $1/z$ is 0.001. The `Z_MULT_FACTOR` has the effect of scaling the $1/z$ values to be slightly larger so that interpolation between the $1/z$ values occurs between larger values and is thus more accurate. The `Z_ADD_FACTOR` has the effect of making all z values slightly smaller before computing $1/z$. The idea is that because of the near z clipping plane, none of the z values will ever be smaller than a certain value. For instance, if the near z plane is at 5, then z will always be at least 5, meaning that $1/z$ is at most 1/5, or 0.2. Even with the `Z_MULT_FACTOR`, there is still a certain range of larger $1/z$ values which never occur due to the near z clipping. The purpose of the `Z_ADD_FACTOR` is to move the clipped z values slightly closer to the origin, so that the range of possible $1/z$ values is larger. These numerical range considerations are particularly important when using fixed-point arithmetic since numerical accuracy is always a troublesome problem—troublesome because we only have a fixed number of bits to store whole and fractional parts, and if our computations are carelessly formulated, the resulting values will require either more or less precision than we have, leading to incorrect results. Essentially, therefore, we use the `Z_MULT_FACTOR` and `Z_ADD_FACTOR` to scale our computations such that the results always lie within the fixed precision of our fixed point values.

The text drawing code in the software rasterizer relies on a table of bitmaps, containing one bitmap for each character, and the two overridden methods `set_text_color` and `draw_text`. The constructor for the rasterizer implementation creates the table of bitmaps in variable `font_bitmaps`. Each character is associated with an 8 by 13 bitmap, stored as a series of 13 bytes. Each bit in each byte of each character's bitmap represents one pixel forming the shape of the character; a value of 1 for the bit means the pixel should be drawn to form the shape of the character, while a value of 0 means the bit should not be drawn. The bits are interpreted such that the most-significant (leftmost) bit is the leftmost pixel to be displayed. Since each bit is 8 bytes (an admittedly hard-coded assumption in the code), one byte represents the eight horizontal pixels belonging to each character's bitmap.

Team-Fly®

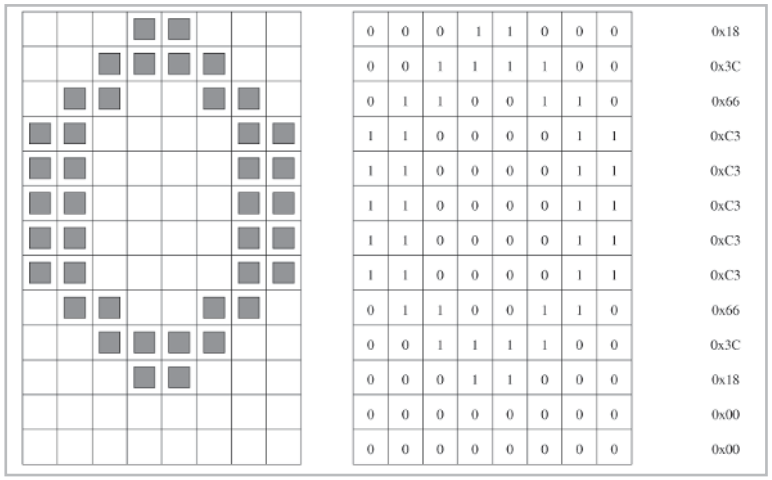


Figure 2-41: The character bitmap for the character "0." Each horizontal line in the bitmap (left) is interpreted as a byte, where a lit pixel represents bit value 1 and an unlit pixel represents bit value 0 (middle). The bits for each byte, represented in hex, are shown on the right, and are the actual values stored in the font bitmap array. We store one such bitmap for each possible character we want to draw (letters, numbers, punctuation, etc.).

We use the following methods to draw the text. Method `set_text_color` takes a red, green, and blue color specification in external format, converts the color to native internal screen format, then stores the color as a series of bytes in the array `text_color`. Then, method `draw_text` draws a given string using the previously set color. It takes each character in the string to be printed, finds the appropriate bitmap for the character by using its ASCII code, and, for each bit in the character's bitmap with a value of 1, draws the color stored in `text_color` into the rasterizer buffer. Thus, the string is drawn character for character, pixel by pixel into the buffer.

A Mesa/OpenGL 3D Rasterizer Implementation: l3d_rasterizer_3d_mesa_imp

Mesa can also draw texture mapped polygons for us. For a computer without hardware acceleration, there is no speed benefit to be gained by using Mesa instead of a custom software texture mapper, because Mesa must do exactly the same math. The picture looks quite different, though, with a 3D graphics card. In this case, Mesa simply forwards the request to draw a texture mapped polygon on to the hardware, which then draws the polygon at speeds impossible to achieve via software alone. Therefore, let's now take a look at a Mesa/OpenGL 3D rasterizer implementation. As usual, the rasterizer implementation is chosen via a factory at the program start, and is plugged into the rasterizer interface. Application code need not be modified to use either software or hardware rasterizers.

Listing 2-12: `ras3_mes.h`

```
#ifndef __RAS3_MES_H
#define __RAS3_MES_H
#include "../tool_os/memman.h"

#include <GL/glut.h>
#include "rast3.h"
#include "ras_mesa.h"

class l3d_rasterizer_3d_mesa_imp :
      virtual public l3d_rasterizer_2d_mesa_imp,
      virtual public l3d_rasterizer_3d_imp
```

```
{
  private:

    GLuint fontOffset;
    void make_raster_font(void);

  protected:
    void texture_setup(const l3d_polygon_3d_textured *p_poly);
    void ogl_texturing_between_glbegin_end
    (const l3d_polygon_3d_textured *p_poly);

    l3d_real ui_xs, ui_ys, ui_c, vi_xs, vi_ys, vi_c, zi_xs, zi_ys, zi_c,
    ui, vi, zi;

    unsigned long text_color;

  public:
    l3d_rasterizer_3d_mesa_imp(int xs, int ys, l3d_screen_info *si);

    void draw_point(int x, int y, unsigned long col);
    void draw_line(int x0, int y0, int x1, int y1, unsigned long col);
    void draw_polygon_flatshaded(const l3d_polygon_2d_flatshaded *p_poly);

    void draw_polygon_textured(const l3d_polygon_3d_textured *p_poly);
    void draw_polygon_textured_lightmapped
    (const l3d_polygon_3d_textured_lightmapped *p_poly);

    void draw_text(int x, int y, const char *string);
    void set_text_color(int red, int green, int blue);

    void clear_buffer(void);
};

class l3d_rasterizer_3d_mesa_imp_factory :
  public l3d_rasterizer_3d_imp_factory {
  public:
    l3d_rasterizer_3d_imp *create(int xs, int ys, l3d_screen_info *si);
};

#endif
```

Listing 2-13: ras3_mes.cc

```
#include "ras3_mes.h"
#include <math.h>
#include "../tool_os/memman.h"

static const GLubyte period[] =
  {0x00, 0x00, 0x18, 0x18, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

static const GLubyte space[] =
  {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

static const GLubyte digits[][13] = {
  {0x0,0x0, 0x18,0x3C,0x66,0xC3,0xC3,0xC3,0xC3,0xC3,0x66,0x3C,0x18},
  {0x0,0x0, 0x7F,0x7F,0xC,0xC,0xC,0xC,0xC,0xC,0x6C,0x3C,0xC},
  {0x0,0x0, 0xFF,0xFF,0xC0,0x60,0x30,0x18,0xC,0x6,0xC6,0x66,0x3C},
  {0x0,0x0, 0x7E,0xC3,0x3,0x6,0xC,0x38,0xC,0x6,0x3,0xC3,0x7E},
  {0x0,0x0, 0x6,0x6,0x6,0x6,0xFF,0xFF,0xC6,0xC6,0x66,0x36,0x1E},
  {0x0,0x0, 0x7E,0xFF,0xC3,0x3,0x3,0xFF,0xFE,0xC0,0xC0,0xC0,0xFE},
  {0x0,0x0, 0x7E,0xFF,0xC3,0xC3,0xC3,0xFF,0xFE,0xC0,0xC0,0xC0,0x7E},
```

```
  {0x0,0x0, 0x60,0x60,0x60,0x60,0x30,0x18,0xC,0x6,0x3,0x3,0xFF},
  {0x0,0x0, 0x7E,0xFF,0xC3,0xC3,0xC3,0x7E,0xC3,0xC3,0xC3,0xC3,0x7E},
  {0x0,0x0, 0x7E,0xFF,0xC3,0x3,0x3,0x7F,0xC3,0xC3,0xC3,0xC3,0x7E}
};

static const GLubyte letters[][13] = {
  {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0x66, 0x3c, 0x18},
  {0x00, 0x00, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
  {0x00, 0x00, 0x7e, 0xe7, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
  {0x00, 0x00, 0xfc, 0xce, 0xc7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc7, 0xce, 0xfc},
  {0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xc0, 0xff},
  {0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xff},
  {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xcf, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
  {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
  {0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x7e},
  {0x00, 0x00, 0x7c, 0xee, 0xc6, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06},
  {0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xe0, 0xf0, 0xd8, 0xcc, 0xc6, 0xc3},
  {0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0},
  {0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xdb, 0xff, 0xff, 0xe7, 0xc3},
  {0x00, 0x00, 0xc7, 0xc7, 0xcf, 0xcf, 0xdf, 0xdb, 0xfb, 0xf3, 0xf3, 0xe3, 0xe3},
  {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xe7, 0x7e},
  {0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
  {0x00, 0x00, 0x3f, 0x6e, 0xdf, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0x66, 0x3c},
  {0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
  {0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0xe0, 0xc0, 0xc0, 0xe7, 0x7e},
  {0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0xff},
  {0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
  {0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
  {0x00, 0x00, 0xc3, 0xe7, 0xff, 0xff, 0xdb, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
  {0x00, 0x00, 0xc3, 0x66, 0x66, 0x3c, 0x3c, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3},
  {0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3},
  {0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x7e, 0x0c, 0x06, 0x03, 0x03, 0xff}
};

l3d_rasterizer_3d_imp * l3d_rasterizer_3d_mesa_imp_factory::create
(int xs, int ys, l3d_screen_info *si)
{
  return new l3d_rasterizer_3d_mesa_imp(xs,ys,si);
}

l3d_rasterizer_3d_mesa_imp::
l3d_rasterizer_3d_mesa_imp(int xs, int ys, l3d_screen_info *si):
    l3d_rasterizer_3d_imp(xs,ys,si),
    l3d_rasterizer_2d_imp(xs,ys,si),
    l3d_rasterizer_2d_mesa_imp(xs,ys,si)
{
  make_raster_font();
  set_text_color(si->ext_max_red,
                 si->ext_max_green,
                 si->ext_max_blue);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(-xsize/2,xsize/2,-ysize/2,ysize/2);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glViewport(0,0,xsize,ysize);

  //- prevent back-facing polygons - those which have a counterclockwise
```

```
  //- vertex orientation - from being drawn, to mimic the behavior of the
  //- software rasterizer.
  glFrontFace(GL_CW);
  glCullFace(GL_BACK);
  glEnable(GL_CULL_FACE);
}

void l3d_rasterizer_3d_mesa_imp::clear_buffer(void)
{
  l3d_rasterizer_2d_mesa_imp::clear_buffer();
}

void l3d_rasterizer_3d_mesa_imp::draw_point(int x, int y, unsigned long col) {
  glBegin(GL_POINTS);
  sinfo->set_color(col);
  glVertex2i(x-(xsize>>1),(ysize>>1)-y);
  glEnd();
}

void l3d_rasterizer_3d_mesa_imp::draw_line(int x0, int y0, int x1, int y1,
    unsigned long col)
{
  glBegin(GL_LINES);
  sinfo->set_color(col);
  glVertex2i(x0-(xsize>>1),(ysize>>1)-y0);
  glVertex2i(x1-(xsize>>1),(ysize>>1)-y1);
  glEnd();
}

void l3d_rasterizer_3d_mesa_imp::draw_polygon_flatshaded
(const l3d_polygon_2d_flatshaded *p_poly)
{
  int i;

  glBegin(GL_POLYGON);
  sinfo->set_color(p_poly->final_color);
  for(i=0; i<p_poly->clip_ivertices->num_items; i++) {
    glVertex2i(iceil((**(p_poly->vlist))
                     [(*p_poly->clip_ivertices)[i].ivertex].transformed.X_)

               - (xsize>>1),

               (ysize>>1) -
               iceil((**(p_poly->vlist))
                     [(*p_poly->clip_ivertices)[i].ivertex].transformed.Y_));
  }
  glEnd();
}

void l3d_rasterizer_3d_mesa_imp::texture_setup
(const l3d_polygon_3d_textured *p_poly)
{

  if (!glIsTexture(p_poly->texture->tex_data->id)) {
    printf("generating texture");
    GLuint glid = (GLuint) (p_poly->texture->tex_data->id);
    glGenTextures(1, & glid );
    p_poly->texture->tex_data->id = glid;

    glBindTexture(GL_TEXTURE_2D, p_poly->texture->tex_data->id);
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
    glEnable(GL_TEXTURE_2D);

    glTexImage2D(GL_TEXTURE_2D,
                 0,
                 GL_RGBA,
                 p_poly->texture->tex_data->width,
                 p_poly->texture->tex_data->height,
                 0,
                 GL_RGBA,
                 GL_UNSIGNED_BYTE,
                 p_poly->texture->tex_data->data);
  }else {
    glEnable(GL_TEXTURE_2D); //- texturing may have been turned off by another

    glBindTexture(GL_TEXTURE_2D,p_poly->texture->tex_data->id);
  }

  float texx[4] = {0.0, 0.0, 1.0, 1.0 };
  float texy[4] = {0.0, 1.0, 1.0, 0.0 };

}

void l3d_rasterizer_3d_mesa_imp::draw_polygon_textured
(const l3d_polygon_3d_textured *p_poly)

{
  texture_setup(p_poly);
  glBegin(GL_POLYGON);
  ogl_texturing_between_glbegin_end(p_poly);
  glEnd();
}

void l3d_rasterizer_3d_mesa_imp::ogl_texturing_between_glbegin_end
(const l3d_polygon_3d_textured *p_poly)
{
  int i;

  GLfloat plane[4];
  l3d_vector plane_normal;

  plane_normal = normalized(p_poly->sfcnormal.transformed
                            - p_poly->center.transformed);

  plane[0] = plane_normal.a[0];
  plane[1] = plane_normal.a[1];
  plane[2] = plane_normal.a[2];
  plane[3] = -plane[0] * p_poly->center.transformed.X_
             - plane[1] * p_poly->center.transformed.Y_
             - plane[2] * p_poly->center.transformed.Z_;

  for(i=0; i<p_poly->clip_ivertices->num_items; i++) {

  #define SCREEN_X ((**(p_poly->vlist)) [(*(p_poly->clip_ivertices))[i].ivertex].transformed.X_)
```

```
  #define SCREEN_Y ((**(p_poly->vlist)) [(*(p_poly->clip_ivertices))[i].ivertex].transformed.Y_)

    l3d_real hf=l3d_mulri(*fovx,*screen_xsize),
             vf=l3d_mulri(*fovy,*screen_ysize);
    float orig_z_denom =
      (plane[0]*(SCREEN_X - xsize/2)/ hf -
       plane[1]*(SCREEN_Y - ysize/2)/ vf +
       plane[2]);
    if(orig_z_denom > -0.0001) {

      //- this would imply that the orig_z value is < 0.0001 ie non-positive
      //- so we cancel the entire polygon drawing operation
      return;
    }
  }

  for(i=0; i<p_poly->clip_ivertices->num_items; i++) {

#define SCREEN_X ((**(p_poly->vlist)) [(*(p_poly->clip_ivertices))[i].ivertex].transformed.X_)
#define SCREEN_Y ((**(p_poly->vlist)) [(*(p_poly->clip_ivertices))[i].ivertex].transformed.Y_)

    glTexCoord2f
    (
      ((l3d_polygon_ivertex_textured *)&(*p_poly->clip_ivertices)[i])
      ->tex_coord.X_,
      ((l3d_polygon_ivertex_textured *)&(*p_poly->clip_ivertices)[i])
      ->tex_coord.Y_
    );

    l3d_real hf=l3d_mulri(*fovx,*screen_xsize),
             vf=l3d_mulri(*fovy,*screen_ysize);
    float orig_z = -plane[3] /
                   (plane[0]*(SCREEN_X - xsize/2)/ hf -
                    plane[1]*(SCREEN_Y - ysize/2)/ vf +
                    plane[2]);

    GLfloat tmp[4];

    tmp[0] =
      (((**(p_poly->vlist))
        [(*(p_poly->clip_ivertices))[i].ivertex].transformed.X_ - xsize/2)

       * orig_z );

    tmp[1] =
      ((-(**(p_poly->vlist))
        [(*(p_poly->clip_ivertices))[i].ivertex].transformed.Y_ + ysize/2)

       * orig_z );
    tmp[2] = 1;
    tmp[3] = orig_z;

    glVertex4fv(tmp);
  }
}

void l3d_rasterizer_3d_mesa_imp::make_raster_font(void)
{
  GLuint i, j;
```

```
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

  fontOffset = glGenLists (128);
  for (i = 0,j = 'A'; i < 26; i++,j++) {
    glNewList(fontOffset + j, GL_COMPILE);
    glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0,
             letters[i]);
    glEndList();
  }

  for (i = 0,j = 'a'; i < 26; i++,j++) {
    glNewList(fontOffset + j, GL_COMPILE);
    glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0,
             letters[i]);
    glEndList();
  }

  for (i = 0,j = '0'; i < 10; i++,j++) {
    glNewList(fontOffset + j, GL_COMPILE);
    glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0,
             digits[i]);
    glEndList();
  }

  glNewList(fontOffset + ' ', GL_COMPILE);
  glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0,
           space);
  glEndList();

  glNewList(fontOffset + '.', GL_COMPILE);
  glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0,
           period);
  glEndList();

}

void l3d_rasterizer_3d_mesa_imp::draw_text(int x, int y, const char *string) {
  sinfo->set_color(text_color);
  glDisable(GL_TEXTURE_2D); //- text should not be displayed using a texture
  glRasterPos2i(x-(xsize>>1),(ysize>>1)-y - 13); //- 13 is height of 1 char

  glPushAttrib (GL_LIST_BIT);
  glListBase(fontOffset);
  glCallLists((GLsizei) strlen(string), GL_UNSIGNED_BYTE, (GLubyte *) string);
  glPopAttrib ();

}

void l3d_rasterizer_3d_mesa_imp::set_text_color(int red, int green, int blue) {
  text_color = sinfo->ext_to_native(red, green, blue);
}

void l3d_rasterizer_3d_mesa_imp::draw_polygon_textured_lightmapped
(const l3d_polygon_3d_textured_lightmapped *p_poly)
{
  GLuint glid = (GLuint) (p_poly->texture->tex_data->id);

  l3d_list<l3d_point> old_texcoords(10);

  //- try to find texture data in surface cache
```

```
if (!p_poly->get_scache()->find_texdata(p_poly->texture->tex_data)) {

  if (glIsTexture(p_poly->texture->tex_data->id)) {

    glDeleteTextures(1, &glid);
  }

  glGenTextures(1, & glid );
  p_poly->texture->tex_data->id = glid;

  glBindTexture(GL_TEXTURE_2D, p_poly->texture->tex_data->id);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
  glEnable(GL_TEXTURE_2D);

  glTexImage2D(GL_TEXTURE_2D,
               0,
               GL_RGBA,
               p_poly->texture->tex_data->width,
               p_poly->texture->tex_data->height,
               0,
               GL_RGBA,
               GL_UNSIGNED_BYTE,
               p_poly->texture->tex_data->data);
}else {
  glBindTexture(GL_TEXTURE_2D,p_poly->texture->tex_data->id);
}

//- based on texture orientation: compute uvz's for each vertex
for(int i=0; i<p_poly->clip_ivertices->num_items; i++) {

  float px =
    ( (**p_poly->vlist) [(*p_poly->clip_ivertices)[i].ivertex].transformed.X_
      - *screen_xsize/2 )
    / (*screen_xsize * *fovx);

  float py =
    (*screen_ysize/2
     -
     (**p_poly->vlist) [(*p_poly->clip_ivertices)[i].ivertex].transformed.Y_ )
    /
    (*screen_ysize * *fovy);

  float ox = p_poly->texture->O.transformed.X_;
  float oy = p_poly->texture->O.transformed.Y_;
  float oz = p_poly->texture->O.transformed.Z_;
  float ux = p_poly->texture->U.transformed.X_ - ox;
  float uy = p_poly->texture->U.transformed.Y_ - oy;
  float uz = p_poly->texture->U.transformed.Z_ - oz;
  float vx = p_poly->texture->V.transformed.X_ - ox;
  float vy = p_poly->texture->V.transformed.Y_ - oy;
  float vz = p_poly->texture->V.transformed.Z_ - oz;

  float u = (px*(oy*vz - oz*vy) + py*(oz*vx - ox*vz) - oy*vx + ox*vy)
            / (px*(uz*vy - uy*vz) + py*(ux*vz - uz*vx) + uy*vx - ux*vy);
```

```
  float v = (px*(oy*uz - oz*uy) + py*(oz*ux - ox*uz) - oy*ux + ox*uy)
            / ((uy*vz - uz*vy)*px + py*(uz*vx - ux*vz) + ux*vy - uy*vx);
  float z = (ox*(uz*vy - uy*vz) + oy*(ux*vz - uz*vx) + oz*(uy*vx - ux*vy))
            / (px*(uz*vy - uy*vz) + py*(ux*vz - uz*vx) + uy*vx - ux*vy);

  old_texcoords[old_texcoords.next_index()] =
    ((l3d_polygon_ivertex_textured *)&(*p_poly->clip_ivertices)[i])
    ->tex_coord;

  ((l3d_polygon_ivertex_textured *)&(*p_poly->clip_ivertices)[i])
  ->tex_coord.set(float_to_l3d_real(u),
                  float_to_l3d_real(v),
                  float_to_l3d_real(z),
                  float_to_l3d_real(1.0));
}

draw_polygon_textured(p_poly);

//- set up mag/min filters to make light maps smoother
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

//- set up blend function to combine light map and text map
glEnable(GL_BLEND);
glBlendFunc(GL_ZERO, GL_SRC_ALPHA);

int i;

//- try to find light map data in cache
glid = (GLuint) (p_poly->lightmap->tex_data->id);
if (!p_poly->get_scache()->find_texdata(p_poly->lightmap->tex_data)) {
  if (glIsTexture(p_poly->lightmap->tex_data->id)) {

    glDeleteTextures(1, &glid);
  }
  printf("generating texture");
  glGenTextures(1, & glid );
  p_poly->lightmap->tex_data->id = glid;
  printf("ID was %d", p_poly->lightmap->tex_data->id);

  glBindTexture(GL_TEXTURE_2D, p_poly->lightmap->tex_data->id);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
  glEnable(GL_TEXTURE_2D);

  glPixelStorei(GL_UNPACK_ALIGNMENT,1);
  glTexImage2D(GL_TEXTURE_2D,
               0,
               GL_RGBA,
               p_poly->lightmap->tex_data->width,
               p_poly->lightmap->tex_data->height,
               0,
               GL_ALPHA,
               GL_UNSIGNED_BYTE,
               p_poly->lightmap->tex_data->data);
```

```
  }else {
    glBindTexture(GL_TEXTURE_2D,p_poly->lightmap->tex_data->id);
  }

  //- restore lumel-space uv's for each clip_ivertex

  for(int i=0; i<p_poly->clip_ivertices->num_items; i++) {
    ((l3d_polygon_ivertex_textured *)&(*p_poly->clip_ivertices)[i])
    ->tex_coord = old_texcoords[i];
  }

  //- draw the same polygon again, but this time with the light map
  glBegin(GL_POLYGON);
  ogl_texturing_between_glbegin_end(p_poly);
  glEnd();

  glDisable(GL_BLEND);

}
```

The class `l3d_rasterizer_3d_mesa_imp` is the 3D rasterizer implementation using Mesa. The first thing to notice is the following line in the constructor:

```
gluOrtho2D(-xsize/2,xsize/2,-ysize/2,ysize/2);
```

This line has the effect of setting the origin of the screen coordinate system to be the center of the screen, not the top left corner as usual. Furthermore, the *y* axis values increase as they move upwards, not downwards; this is the normal *y* axis orientation, not the reversed *y* orientation. The reason we must make this change is for the texture mapping code, as we see shortly.

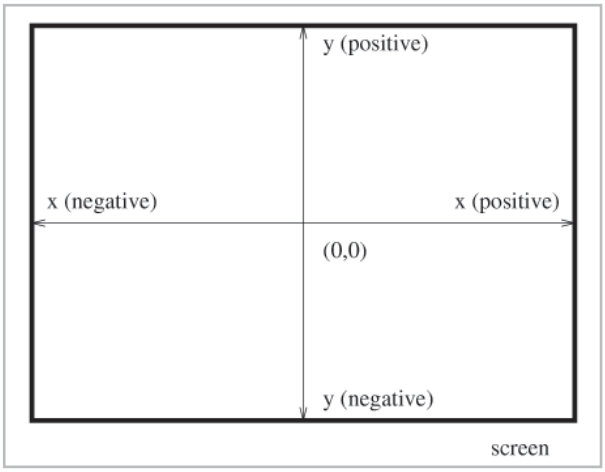


Figure 2-42: The screen coordinate system for the Mesa 3D rasterizer implementation class. Notice that the origin is at the center of the screen, and that the y axis orientation is normal.

Because we have redefined the origin and *y* axis orientation, we must rewrite all of the inherited drawing routines from the 2D rasterizer to use the new coordinate system. The `draw_point`, `draw_line`, and `draw_polygon_flatshaded` routines perform the same function and call the same OpenGL routines as in the parent class `l3d_rasterizer_2d_mesa_imp`, only here in the 3D rasterizer we offset all of the incoming *x* and *y* coordinates to use the new screen-centered coordinate system.

Team-Fly®

Like the software 3D rasterizer, the Mesa 3D rasterizer also overrides the routines `set_text_color` and `draw_text` to draw text into the rasterizer buffer, although here we must use OpenGL commands to draw text. Method `set_text_color` converts the given external red, green, and blue color into the native pixel format, and stores it in the variable `text_color`. Method `draw_text` then calls the `set_color` routine of the screen info object with the color `text_color`, which for Mesa calls `glColor4f`. Then, after setting the color, method `draw_text` disables texture mapping, since otherwise text might be drawn using the currently active texture map if texture mapping had been enabled earlier, and sets the raster position at which the text is to be drawn (in 2D screen coordinates, with the screen-centered coordinate system) with `glRasterPos2i`. Finally, the text is drawn with the calls to `glPushAttrib`, `glListBase`, `glCallLists`, and `glPopAttrib`. The calls to `glPushAttrib` and `glPopAttrib` save and restore the OpenGL state variables before and after the calls to `glListBase` and `glCallLists`. The calls to `glListBase` and `glCallLists` execute a *display list*. A display list is an OpenGL-specific concept; it is a collection of OpenGL commands that can be executed as a logical unit, something like a subroutine consisting of OpenGL commands. Each display list is assigned an integer identifier by OpenGL. Execution of a display list may be faster than individually calling the separate OpenGL commands, because the commands in the display list may be cached on or optimized by the graphics hardware. For the case of drawing text, we use a separate display list for each character, created in the initialization routine `make_raster_font`. This routine first allocates space for 128 consecutive display lists with `glGenLists`, which returns the integer identifier for the first of the lists. Then, for each display list corresponding to a character we wish to display, we enter commands into the display list by executing `glNewList`, which begins the list, `glBitmap`, which draws the bitmap data to the screen using the current color, and `glEndList` to finish the list.

The method `draw_polygon_textured` is the new routine of interest which uses OpenGL calls to draw a texture mapped polygon, using hardware acceleration if present. It calls `texture_setup` to initialize the texture, then calls `glBegin(GL_POLYGON)` to draw a polygon, and calls `ogl_texturing_between_glbegin_end` to actually issue the OpenGL texture mapping commands, finishing with a call to `glEnd`.

The method `texture_setup` has the responsibility of making the texture image data available to OpenGL. Recall, OpenGL is a state machine; issued commands are affected based on the currently set parameters. The texture is one such parameter; to draw a texture mapped polygon, we must first select the texture. OpenGL (version 1.1 and later) stores textures in *texture objects*. A texture object is referenced by a non-zero integer identifier, which we store in the `id` member of the `l3d_texture_data` class. In method `texture_setup`, we first check to see if the OpenGL texture data has already been allocated and initialized via `glIsTexture`, passing the texture `id`. If the texture does exist, we simply make it the current texture for the subsequent drawing commands, through the call to `glBindTexture`. If the texture does not exist, we must create it for OpenGL. First, we call `glGenTextures` to return a new, unused texture `id` number, and store it with the texture data. We then call `glBindTexture` to make this texture object current.

At this point we have created and bound a new texture to OpenGL, but the texture is still empty. Therefore, we issue additional commands to make the texture image data known to OpenGL. First, we call `glTexParameteri` several times to set up some parameters for the texture. The first two calls to `glTexParameteri` cause scaled-up (magnified) or scaled-down (minified) textures to be drawn using the closest pixel within the texture map; the alternative would be to use some sort of average of several pixels within the texture map, which produces a smoother result but is slower. The second two calls to `glTexParameteri` cause the texture coordinates to wrap around or repeat outside of the (0,0) to (1,1) square in texture space. Then, we call `glHint` with the parameters GL_PERSPECTIVE_CORRECTION_HINT and GL_NICEST, which causes OpenGL to try to perform the "nicest" perspective correction possible when doing texture mapping. The *perspective correction* is simply the interpolation of u/z, v/z, and $1/z$ with a division to retrieve the mathematically correct u, v, and z values, as we already have been doing; texture mapping without perspective correction interpolates the u, v, and z values directly (which we did in software for short runs of 16 pixels) and is faster but leads to incorrect and visually disturbing artifacts. The call to `glEnable(GL_TEXTURE_2D)` turns on the texture mapping in OpenGL.



TIP If you want to see the effect of texture mapping without perspective correction in the software rasterizer, try setting the constant `LINEAR_RUN_SHIFT` in class `l3d_rasterizer_3d_sw_imp` to be 16. This means that each linear run is 2 ^ 16 or 65,536 pixels long, meaning that across an entire scanline, the u and v values (instead of u/z, v/z, and $1/z$, which would be correct) are linearly interpolated. This results in distortion and warping of the texture image but faster performance because no division operations are performed within the scanline.

Finally, and most importantly, we call `glTexImage2D`, which actually sends the texture image data to OpenGL. The first parameter specifies that we are working with a 2D texture. The next parameter is only used with multiple levels of detail, where a different texture is drawn depending on the distance from the camera, so we set it to 0. The third parameter, which we set to GL_RGBA, means that our texture data specifies red, green, blue, and also possibly alpha (transparency) components for our colors. The next two parameters are the width and height of the texture; after that comes an optional border parameter to surround the texture, which we set to 0 (no border). The next two parameters specify the format of the texture image data: red, green, blue, and alpha components, each as a byte—in other words, 32-bit RGBA, which is also the same pixel format specified in the Mesa screen information class `l3d_screen_info_rgb_mesa`. The final parameter is a pointer to the actual texture image data itself.

The method `ogl_texturing_between_glbegin_glend` issues the actual OpenGL commands to draw the textured polygon. This routine must be called between `glBegin(GL_POLYGON)` and `glEnd`, as the name of the routine implies. Fundamentally, the routine works as follows: for each vertex in the polygon, we call `glTexCoord2f` with the (u,v) coordinates of the polygon, which we stored in the `l3d_polygon_ivertex_textured.tex_coord` member. Then, we call `glVertex4fv` with the spatial coordinates of the vertex. After specifying the texture and spatial coordinates for all vertices of the polygon, the routine returns, `glEnd` is called, and the texture mapped polygon is then drawn by OpenGL.

There is, however, one tricky computation which must be performed within the method `ogl_texturing_between_glbegin_glend`. The problem is that we must pass a pre-projection vertex to Mesa, and must allow Mesa to perform the perspective division, in order for the texture mapping to occur correctly. We cause Mesa to do perspective division by using homogeneous coordinates. The last operation performed by the hardware on a coordinate is the division by the homogeneous w component. If we set w to be the 3D z component, this division effects the perspective projection. Our problem is that we already performed the perspective division, in `l3d_object::apply_perspective_projection`, and we already know exactly which 2D pixel location we need. So our task is, from this already known 2D pixel location, to compute the pre-projection 3D coordinate which, when re-projected by Mesa, yields exactly the 2D pixel location we desire. This is a form of "reverse projection," very similar to the way that we "reverse projected" from screen space into texture space. Here, we must reverse project from 2D screen space back into 3D world space.



CAUTION You may be tempted to believe that we can simply save the vertex coordinates before perspective projection, in one of the transformed intermediates within the vertex. Then, you might think, later on within the Mesa rasterizer, we would simply retrieve the pre-projection coordinate and pass it to Mesa. The flaw with this reasoning is that after projection, we perform an additional clipping operation in 2D, clipping the projected polygon to the view window. This clipping can produce additional vertices, which did not directly result from a 3D to 2D projection operation. For such vertices produced by a 2D clip, we have no *a priori* knowledge of the corresponding 3D point. Therefore, we cannot just save the pre-projection 3D coordinate ahead of time; we must dynamically compute it later as needed.

This reverse projection from screen space into world space is quite easy and relies on the plane equation of the polygon which we developed earlier. Consider the following system of equations:

Equation 2-35
$$\begin{aligned} ax + by + cz + d &= 0 \\ x_p &= \frac{x}{z} \\ y_p &= \frac{y}{z} \end{aligned}$$

This is a system of three equations. The first is the plane equation of the polygon, and it constrains how the x, y, and z values in 3D relate to one another: they must lie on a plane. The a, b, and c constants of the plane equation, recall, are simply the normalized x, y, and z components of the surface normal to the plane; the d component can be calculated by substituting a known x, y, and z point on the plane (such as the polygon center) into the plane equation and solving for d. The second two equations are the simplified perspective projection equations ignoring field of view and y axis reversal; as seen earlier, we can easily convert from these simple projected coordinates to the real screen projected coordinates.

Solving this system for x, y, and z (by using Calc) yields the following system of equations.

Equation 2-36
$$\begin{aligned} x &= -\frac{d}{a\,x_p + b\,y_p + c} x_p = z\,x_p \\ y &= -\frac{d}{a\,x_p + b\,y_p + c} y_p = z\,y_p \\ z &= -\frac{d}{a\,x_p + b\,y_p + c} \end{aligned}$$

These equations give us the world space coordinates from the screen space coordinates—a "reverse projection."

As you can see here, it is very important to express a problem in terms of known and unknown quantities. If we are only given the perspective projection equations relating x, y, and z, we do not have enough information to perform reverse projection. Mathematically, we have two equations ($x=x/z$, $y=y/z$) but three unknowns (x, y, and z). Geometrically, the perspective projected point could have come from any point in 3D which lies on the light ray. But, by adding the plane equation to the system of equations, we do have enough information to perform reverse projection. Mathematically, we now have three equations and three unknowns. Geometrically, the additional constraint of the plane equation means that the point in 3D must not only project to the given 2D point, but it must also lie on the plane of the polygon. The general observation here is that an additional equation relating the known quantities allowed us to solve the system of equations. The challenge is to determine which known information might be of use in finding some unknown quantities.

So, to use these equations, we first convert from real screen projected coordinates to the simple projected coordinates, using the earlier Equation 2-30. Notice that this conversion relies on the horizontal and vertical field of view terms and screen size, which is why the `l3d_texture_computer` class, from which the Mesa rasterizer inherits, stores these values.



CAUTION The fact that the Mesa texture mapping code requires the horizontal and vertical field of view terms and the screen size means that you must initialize these values before attempting to perform any texture mapping operations.

The following sample program `textest` initializes these field of view terms in the beginning of the world constructor. The reason we must perform this initialization is that in the rasterizer we store *pointers* to the actual variables storing the field of view and screen size terms. This allows us to change values in the external variables (e.g., to interactively change the field of view) without needing also to propagate the updated values to the rasterizer. But the fact that the rasterizer stores pointers to the variables means that failure to initialize these pointers followed by an attempt to perform Mesa texture mapping results in an attempt to access memory through an uninitialized pointer—a practice which, to put it mildly, is best avoided. Ordinarily, such initialization would be done by including a required parameter in the class's constructor, so that it is not possible to create an object incorrectly, but in this case we create the rasterizer implementation indirectly through a factory, where the factory's original creation interface (defined in the introductory companion book *Linux 3D Graphics Programming*) was defined before we even saw the notion of field of view. Since the code philosophy in this book emphasizes a bottom-up approach, we'll have to live with this small inconvenience for now. In a real production 3D program, you probably would have

no need to create 2D rasterizer implementations, only 3D rasterizer implementations, meaning that you could just change the factory creation interface and break the old code creating 2D rasterizer implementations. However, for pedagogical purposes, in this book we retain backward compatibility with the code we developed earlier.

Then, having obtained the simple projected coordinates, we use the above equations to determine the 3D pre-projection coordinates. We finally pass these pre-projection 3D coordinates on to OpenGL in the `glVertex4fv` command. The `glVertex4fv` function expects as a parameter an array in which four x, y, z, and w components have been stored. We must specify the 3D z coordinate as the fourth (homogeneous w) component, so that Mesa has the opportunity to perform the homogeneous division by w, thus effecting the perspective division by z and therefore allowing perspective correct texture mapping through OpenGL. It is also important that we not pass the 3D z coordinate as the third (3D z) component to `glVertex4fv`, instead passing the constant 1. What does this achieve? Remember that OpenGL divides all components by the w component. This means that, after the division by w, which we set equal to the 3D z value, the third component of the OpenGL vertex contains $1/z$. As we see in Chapter 4, this $1/z$ value can be used for the z buffer algorithm.



NOTE Note that the last operation Mesa performs on the coordinate is the division by the homogeneous component w. This is the reason that, for the Mesa 3D rasterizer, we redefined the origin of our screen coordinate system to be the center of the screen with a normal y axis orientation. Since the division by w is the last operation, we have no way of telling Mesa "oh, and after the homogeneous division, please also offset the x and y values by half the screen size and reverse the y axis orientation." If we want Mesa to perform the perspective division—which should be the case for texture mapping to occur correctly—then we have no control over the post-projection coordinates, since they are taken over by the hardware at that point.

Sample Program: textest

Congratulations, you now understand basic texture mapping! Let's now take a look at a sample program using texture mapping, so that all of this mind-boggling math actually produces a visual result. With an understanding of the basic concepts of texture mapping, using the texture mapping classes is comparatively easy. The next sample program, `textest`, creates a navigable scene with two texture mapped pyramids. One of the pyramids rotates, to provide some animation in the scene. You can fly through the scene with the normal navigation keys of the `l3d_pipeline_world` class.

The code is in Listings 2-14 through 2-18. File `shapes.cc` is the most significant file, which loads textures and creates texture-mapped polygons.

Listing 2-14: `shapes.h`, the textured object class declaration for program `textest`

```
#include "../lib/geom/object/object3d.h"
#include "../lib/geom/polygon/p3_tex.h"
#include "../lib/geom/texture/texture.h"
#include "../lib/geom/texture/texload.h"

class pyramid:public l3d_object {
    int xtheta;
```

```
  public:
    int dx_theta;
    pyramid(l3d_texture_loader *l);
    virtual ~pyramid(void);
    /* virtual */ int update(void);
};
```

Listing 2-15: shapes.cc, the textured object class implementation for program textest

```
#include "shapes.h"
#include "../lib/tool_os/memman.h"
#include <stdlib.h>
#include <string.h>

pyramid::pyramid(l3d_texture_loader *l) :
    l3d_object(4)
{
  (*vertices)[0].original.set(float_to_l3d_real(0.),float_to_l3d_real(0.),float_to_l3d_real(0.),
          float_to_l3d_real(1.));
  (*vertices)[1].original.set(float_to_l3d_real(10.0),float_to_l3d_real(0.),float_to_l3d_real(0.),
          float_to_l3d_real(1.));
  (*vertices)[2].original.set(float_to_l3d_real(0.),float_to_l3d_real(10.),float_to_l3d_real(0.),
          float_to_l3d_real(1.));
  (*vertices)[3].original.set(float_to_l3d_real(0.),float_to_l3d_real(0.),float_to_l3d_real(10.),
          float_to_l3d_real(1.));

  l3d_polygon_3d_textured *p;
  int pi;

  pi = polygons.next_index();
  polygons[pi] = p = new l3d_polygon_3d_textured(3);

  polygons[pi]->vlist = &vertices;
  (*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=0;
  (*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=1;
  (*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=3;

  l->load("test.ppm");
  p->texture = new l3d_texture;
  p->texture->tex_data = new l3d_texture_data;
  p->texture->tex_data->width = l->width;
  p->texture->tex_data->height = l->height;
  p->texture->tex_data->data = l->data;
  p->texture->owns_tex_data = true;
  p->texture->O = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[0].ivertex];
  p->texture->U = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[1].ivertex];
  p->texture->V = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[2].ivertex];
  polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();
  p->assign_tex_coords_from_tex_space(*p->texture);

  pi = polygons.next_index();
  polygons[pi] = p = new l3d_polygon_3d_textured(3);
  polygons[pi]->vlist = &vertices;

  (*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=2;
  (*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=3;
  (*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=1;

  p->texture = new l3d_texture;
  p->texture->tex_data = new l3d_texture_data;
  p->texture->tex_data->width = l->width;
```

```
p->texture->tex_data->height = l->height;
p->texture->tex_data->data = l->data;
p->texture->owns_tex_data = false;

p->texture->O = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[0].ivertex];
p->texture->U = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[1].ivertex];
p->texture->V = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[2].ivertex];

polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();
p->assign_tex_coords_from_tex_space(*p->texture);

pi = polygons.next_index();
polygons[pi] = p = new l3d_polygon_3d_textured(3);
polygons[pi]->vlist = &vertices;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=0;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=2;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=1;

p->texture = new l3d_texture;
p->texture->tex_data = new l3d_texture_data;
p->texture->tex_data->width = l->width;
p->texture->tex_data->height = l->height;
p->texture->tex_data->data = l->data;
p->texture->owns_tex_data = false;

p->texture->O = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[0].ivertex];
p->texture->U = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[1].ivertex];
p->texture->V = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[2].ivertex];

polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();
p->assign_tex_coords_from_tex_space(*p->texture);

pi = polygons.next_index();
polygons[pi] = p = new l3d_polygon_3d_textured(3);
polygons[pi]->vlist = &vertices;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=3;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=2;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=0;

p->texture = new l3d_texture;
p->texture->tex_data = new l3d_texture_data;
p->texture->tex_data->width = l->width;
p->texture->tex_data->height = l->height;
p->texture->tex_data->data = l->data;
p->texture->owns_tex_data = false;

p->texture->O = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[0].ivertex];
p->texture->U = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[1].ivertex];
p->texture->V = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[2].ivertex];

polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();
p->assign_tex_coords_from_tex_space(*p->texture);

num_xforms = 2;
xtheta=0;
modeling_xforms[0] = l3d_mat_rotx(xtheta);
modeling_xforms[1].set
( float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.),
  float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.),
  float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.),
```

```
    float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.) );

  modeling_xform=
    modeling_xforms[1] | modeling_xforms[0];
}

pyramid::~pyramid(void) {
}

int pyramid::update(void) {
  xtheta += dx_theta; if (xtheta>359) xtheta-=360;
  modeling_xforms[0]=l3d_mat_rotx(xtheta);
  modeling_xform=
    modeling_xforms[1] | modeling_xforms[0];
}
```

Listing 2-16: `textest.h`, the custom world class declaration for program `textest`

```
#ifndef __TEXTEST_H
#define __TEXTEST_H

#include "../lib/geom/world/world.h"
#include "../lib/pipeline/pi_wor.h"
#include "../lib/geom/texture/texture.h"
#include "../lib/geom/texture/tl_ppm.h"

class my_world:public l3d_world {
  public:
    my_world(void);
    void update_all(void);

    virtual ~my_world(void) {delete texture_loader; }
    l3d_texture_loader *texture_loader;
};

#endif
```

Listing 2-17: `textest.cc`, the custom world class implementation for program `textest`

```
#include "textest.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdarg.h>

#include "../lib/geom/object/object3d.h"
#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rasteriz.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/geom/world/world.h"
#include "../lib/geom/texture/texture.h"
#include "../lib/geom/texture/tl_ppm.h"

#include "shapes.h"

void my_world::update_all(void) {
  l3d_world::update_all();
}
```

```
my_world::my_world(void)
    : l3d_world(400,300)
{
  rasterizer_3d_imp->fovx = &(camera->fovx);
  rasterizer_3d_imp->fovy = &(camera->fovy);
  rasterizer_3d_imp->screen_xsize = &(screen->xsize);
  rasterizer_3d_imp->screen_ysize = &(screen->ysize);

  l3d_screen_info *si = screen->sinfo;
  texture_loader = new l3d_texture_loader_ppm(screen->sinfo);

  camera->VRP.set(0,0,-50,0);
  camera->near_z = float_to_l3d_real(5.5);
  camera->far_z = int_to_l3d_real(500);

  int i,j,k,onum;

  i=10; j=0; k=20;
  k=0;

  for(i=1; i<10; i+=5) {
    pyramid *p;
    objects[onum = objects.next_index()] = p = new pyramid(texture_loader);
    p->dx_theta = i-1;

    if (objects[onum]==NULL) exit;
    objects[onum]->modeling_xforms[1].set
    ( int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(-i),
      int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(i/2),
      int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(i),
      int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1) );
    objects[onum]->modeling_xform =
      objects[onum]->modeling_xforms[1] |
      objects[onum]->modeling_xforms[0] ;
    onum++;
  }

  screen->refresh_palette();
  screen->sinfo->compute_light_table();
  screen->sinfo->compute_fog_table();
}
```

Listing 2-18: `main.cc`, the main program file for program `textest`

```
#include "textest.h"
#include "../lib/system/fact0_2.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdarg.h>

main() {
  l3d_dispatcher *d;
  l3d_pipeline_world *p;
  my_world *w;

  factory_manager_v_0_2.choose_factories();
  d = factory_manager_v_0_2.dispatcher_factory->create();
```

```
    w = new my_world();
    p  = new l3d_pipeline_world(w);
    d->pipeline = p;
    d->event_source = w->screen;

    d->start();

    delete p;
    delete d;
    delete w;
}
```

The main program file, `main.cc`, begins by choosing the factories using the new `l3d_factory_manager_v_0_2` class. This class chooses the 3D rasterizer implementation factory: either software or OpenGL/Mesa. Then, the main program simply creates the world, pipeline, and dispatcher objects, hooks them together, and starts the dispatcher.

The class `my_world` is our custom world class. First, it stores pointers to the field of view and screen size variables into the 3D rasterizer, since in the case of Mesa rasterization we need these variables to perform the reverse projection (as mentioned earlier). Next, the class creates a texture loader, of type `l3d_texture_loader_ppm`, which is used by the `pyramid` class (covered in the next paragraph) to load the texture images from disk. Class `my_world` then populates the world with a number of objects of type `pyramid`.

The class `pyramid` is the real focus of interest for this program. Class `pyramid`, inheriting from class `l3d_object`, is a texture mapped, rotating pyramid. The member variable `xtheta` is the current rotation angle; `dx_theta` is the change in `xtheta` per unit time. The constructor requires a parameter of type `l3d_texture_loader` so that the pyramid can load the texture image data from disk. In the constructor, we first define a vertex list, as usual. Then, we create a number of polygons referencing the vertex list, but of the new type `l3d_polygon_3d_textured`.

```
pi = polygons.next_index();
polygons[pi] = p = new l3d_polygon_3d_textured(3);
```

After creating the polygon and assigning its vertex indices as usual, we then load a texture image from disk by using the texture loader, then assign the newly loaded texture to the polygon. The texture image is a 64 by 64 pixel color image saved in PPM format. Since the polygon is the only one referencing the texture, it is responsible for freeing the data—hence, the `owns_tex_data` member is set to true.

```
l->load("test.ppm");
p->texture = new l3d_texture;
p->texture->tex_data = new l3d_texture_data;
p->texture->tex_data->width = l->width;
p->texture->tex_data->height = l->height;
p->texture->tex_data->data = l->data;
p->texture->owns_tex_data = true;
```

Next, we have to define the orientation of the texture space, which places the texture in 3D space relative to the polygon. In this case, we directly use the first vertex of the polygon as the texture space origin, and the two edges of the polygon as the texture space axes. This has the effect of causing the texture image to lie exactly on the face of the polygon.

Team-Fly®

```
p->texture->O = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[0].ivertex];
p->texture->U = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[1].ivertex];
p->texture->V = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[2].ivertex];
```

Finally, we perform some last necessary computations in 3D. We compute the center and surface normal of the polygon, and finally, based on the previous texture space definition, we assign (u,v) texture coordinates to each of the polygon's vertices by using the world to texture space matrix.

```
polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();
p->assign_tex_coords_from_tex_space(*p->texture);
```

Creating and initializing the texture mapped polygons in this manner is actually all we need to do in order to make use of texture mapped polygons in our programs. Having already chosen a 3D rasterizer implementation at program start that is capable of drawing texture mapped polygons, we are free to create texture mapped polygons and assign them to our 3D objects. All routines that draw or otherwise manipulate 3D polygons do so through abstract pointers and virtual function calls, meaning that the texture mapped polygons can be used with no impact to existing code. They are finally displayed through a virtual function call to `draw`, which then ends up calling the rasterizer implementation to draw the texture mapped polygon on-screen.

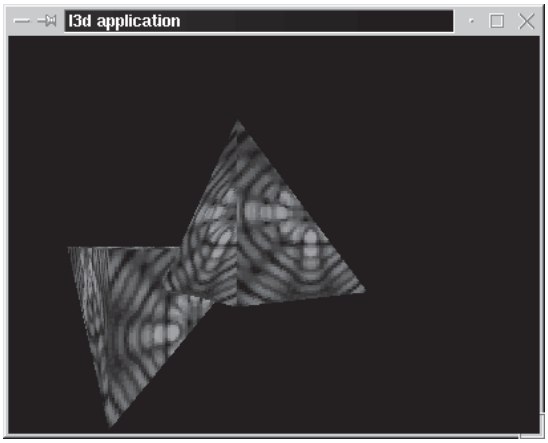


Figure 2-43: Output from program `textest`, displaying two texture mapped pyramid objects.

With texture mapping under our belts, it is now time to return to the topic of light mapping, which builds on the topic of texture mapping.

Light Mapping Revisited

Recall, light mapping is a means of applying computed light intensities to a polygon. It works by defining a grid of lumels on top of the polygon; each lumel in the lumel grid may be assigned a separate light intensity.

We mentioned earlier that the main question with light mapping is the determination of which lumel in the light map corresponds to the current pixel. We can now recognize this as a texture mapping problem: to solve it, we define and orient the light map exactly as we defined the texture map, by means of an additional coordinate system. Then we use the texture mapping equations to convert between screen space, world space, and texture (in this case, light map) space.

The difference between texture mapping and light mapping is that with texture mapping, the color in the texture map defines the final color to be displayed on-screen; with light mapping, the light intensity in the light map does not define the color to be displayed, but instead affects an existing color, making it either brighter or darker via a lighting table.

There are a few ways to implement light mapping. One way would be to modify the inner loop of the rasterizer to perform a double texture mapping calculation (once for the texture map, once for the light map), then to use the lighting table to find the value of the lighted color and display it. This requires rather invasive changes to the rasterization code, since we now have to handle both a texture map and a light map. We don't explore this option further here.

Another way to implement light mapping is to introduce the concept of a *surface*, which is nothing more than a precombined texture map and light map. By precombining the texture map and light map into a surface, we can treat the new surface as a single texture, and associate this single texture with the polygon, which can then be rasterized using the exact same texture mapping routine as before. We'll look at this solution in more detail shortly.

A third, and perhaps simplest, way of realizing light mapping is to perform two entire rasterization passes for each polygon. First, we rasterize the polygon with texture but no lighting. Then, we re-rasterize the same polygon, but using the light map instead of the texture map. During rasterization, instead of drawing the colors of the light map (which are all just various intensities of white), we alter the existing, already drawn colors from the texture map. The end result is the colors of the texture map lighted by the intensities in the light map. Since this strategy requires two entire rasterization passes, it is only really practical in the presence of hardware acceleration. We also look at this solution in the coming sections.

Software Light Mapping

Let's begin by looking at a technique for implementing light mapping in software. The method relies on surfaces and a surface cache.

Surfaces

A *surface* is a pre-tiled, pre-lit texture. A texture and a light map combine to form a surface; the surface can then be treated as a normal texture. With this approach, light mapping becomes simply a sort of preprocessing step before drawing the texture mapped polygons. In theory, this sounds easy enough: for each texel in the texture map, simply light it according to the corresponding lumel in the light map, and we're done, right? In practice, several problems prevent such a straightforward implementation. First of all, the texture map and the light map have different sizes. Typically the light map will have a lower resolution than the texture map because a very fine resolution of lighting detail is not always needed. Secondly, and this is where it becomes tricky, the texture may be scaled. If we have scaled the texture down, this means that the texture then repeats itself (we also say *tiles* itself) across the surface of the polygon. The problem is that although the texels of the texture map repeat themselves across the surface of the polygon, the lumels of the light map must not repeat themselves across the surface of the polygon. A few moments of thought make this clear: if the lumels were also repeated across the polygon, then altering the intensity of one lumel would alter the intensity of several points on the polygon, which

is not at all what is desired! Instead, each lumel should individually control the intensity of one part of the polygon, regardless of whether the texture underneath repeats itself or not. See Figure 2-44.

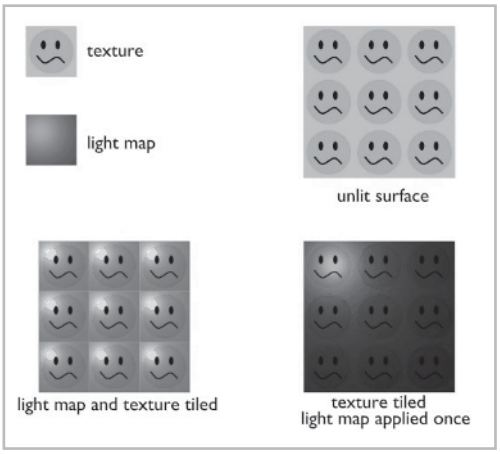


Figure 2-44: We can tile the texture, but we cannot tile the light map.

The solution to this quandary is to create a surface which may be larger than the original texture. Specifically, the surface must be exactly large enough to contain the number of repetitions of the texture occurring on the polygon; the exact number of repetitions depends on the relative sizes and orientations of the polygon and the texture. This is what we mean when we say that a surface is a "pre-tiled" texture. Then, we apply the light map to this larger, repeated texture. The result is a single, large texture image, which (a) contains one or more repetitions of the original texture image, and (b) has been illuminated by the light map.

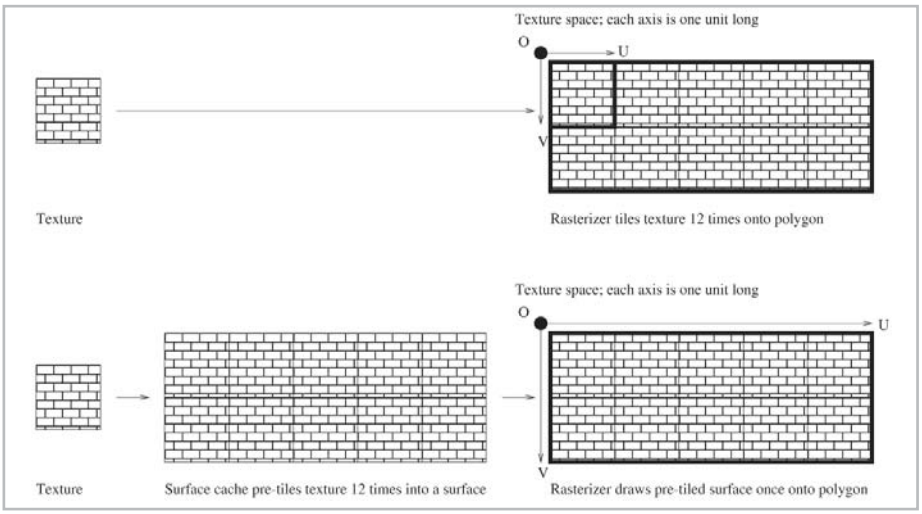


Figure 2-45: Texture mapping without light mapping tiles the texture within the rasterizer. Texture mapping with light mapping tiles the texture beforehand into a surface, which is then treated as one large texture by the rasterizer.

After creating the pre-tiled, pre-lit surface (which may be larger than the original texture), we must then redefine the texture coordinates of the associated polygon. If the surface is larger than the original texture, the new texture coordinates must be appropriately smaller. For instance, if the polygon contains exactly two repetitions of the original texture horizontally, then the original *u* values go from 0 to 2. The larger surface is then twice as wide as the original texture and already contains two repetitions of the original texture, which means that only one repetition of the surface should be mapped to the polygon, meaning that the new *u* values should only go from 0 to 1.

We create surfaces "on demand"—as soon as we attempt to draw a polygon, we first of all see if we have created a surface for the polygon. Each polygon has a unique surface. If the polygon's surface has already been created beforehand, we simply reuse the previous surface and draw the polygon with the surface as the texture. If the polygon's surface has not been created, we combine the texture and light maps into the surface and store the surface for future use. A surface is therefore created as soon as it is needed to be drawn. (A predictive algorithm could also attempt to create surfaces for polygons which are not yet visible but which might soon be visible, to improve performance.) This all implies that we need some means of managing which surfaces have been created and which have not. This is the role of the *surface cache*, which determines for a particular polygon if its surface has already been created or not. The surface cache can also discard old surfaces which have not been used in a long time to save memory.

Let's now look at the l3d classes for light mapping. The class `l3d_surface` is a surface. Member variable `polygon` points to the polygon with which the surface is associated. Member variable `surface` is a pointer to an object of type `l3d_texture`—after all, a surface is nothing more than a texture which has been pre-tiled and pre-lit. Member variable `age` keeps track of the relative age of the surface, so that older surfaces can be discarded to make space for newer surfaces. Of course, if a discarded surface is then referenced again, it must be regenerated.

Listing 2-19: `scache.h`

```
#ifndef __SCACHE_H
#define __SCACHE_H
#include "../../tool_os/memman.h"

class l3d_texture;
class l3d_texture_data;

#include "../../tool_2d/scrinfo.h"
#include "../../datastr/list.h"

class l3d_polygon_3d_textured_lightmapped;

class l3d_surface {
  public:

    l3d_surface(void);
    const l3d_polygon_3d_textured_lightmapped *polygon;
    l3d_texture *surface;

    int age;

    virtual ~l3d_surface(void);
};
```

```
class l3d_surface_cache {
  protected:
    l3d_screen_info *sinfo;
    l3d_list<l3d_surface> *surfaces;

  public:
    l3d_surface_cache(l3d_screen_info *sinfo) {
      surfaces = new l3d_list<l3d_surface> (10);
      this->sinfo = sinfo;
    }

    virtual ~l3d_surface_cache(void) {
      int i;
      delete surfaces;
    }

    l3d_texture *combine_lightmap_and_texmap(int width,
        int height,
        const l3d_polygon_3d_textured_lightmapped *polygon);

    void reset(void) {
      delete surfaces;
      surfaces = new l3d_list<l3d_surface> (10);
    }

    l3d_texture_data *find_texdata(l3d_texture_data *texdata);
};

#include "../polygon/p3_tex.h"
#include "../polygon/p3_ltex.h"

#endif
```

Listing 2-20: scache.cc

```
#include "scache.h"

#include "../../tool_2d/si_rgb.h"
#include "../../tool_os/memman.h"

l3d_surface::l3d_surface(void) {
  surface = NULL;
  polygon = NULL;
}

l3d_surface::~l3d_surface(void)
{

  if(surface) delete surface;
if(polygon) {polygon->surface = NULL; }
}

l3d_texture *l3d_surface_cache::
combine_lightmap_and_texmap(int width,
                            int height,
                            const l3d_polygon_3d_textured_lightmapped *polygon)
{

  {
    int i;
```

```
  for(i=0; i<surfaces->num_items; i++) {
    if((*surfaces)[i].polygon == polygon) {
      return (*surfaces)[i].surface;
    }
  }
}

l3d_texture *new_tex;
l3d_surface *new_surface;

new_tex = new l3d_texture;

int next_surface_index = surfaces->next_index();
(*surfaces)[next_surface_index].polygon = polygon;
(*surfaces)[next_surface_index].surface = new_tex;

new_tex->tex_data = new l3d_texture_data;
new_tex->owns_tex_data = true;
new_tex->tex_data->width = width;
new_tex->tex_data->height = height;
new_tex->tex_data->data = new unsigned char [sinfo->bytes_per_pixel *
                          new_tex->tex_data->width *
                          new_tex->tex_data->height];

int i,j,u,v;

//- tile texture into larger tiled area
for(v=0; v<new_tex->tex_data->height; v++) {
  for(u=0; u<new_tex->tex_data->width; u++) {
    for(i=0; i<sinfo->bytes_per_pixel; i++) {
      *(new_tex->tex_data->data
        + (v*new_tex->tex_data->width + u) * sinfo->bytes_per_pixel
        + i)
      =
        *(polygon->texture->tex_data->data
          + ( ((v&(polygon->texture->tex_data->height-1))*polygon->texture->tex_data->width
               + (u&(polygon->texture->tex_data->width-1))) * sinfo->bytes_per_pixel
              + i ) ) ;
    }
  }
}

unsigned char *c2 = new_tex->tex_data->data;
unsigned char *l = polygon->lightmap->tex_data->data;
l3d_real lx, ly, dlx, dly;

dly = l3d_divrr(int_to_l3d_real(polygon->lightmap->tex_data->height) ,
                int_to_l3d_real(new_tex->tex_data->height));
dlx = l3d_divrr(int_to_l3d_real(polygon->lightmap->tex_data->width) ,
                int_to_l3d_real(new_tex->tex_data->width));
for(j=0, ly=int_to_l3d_real(0); j<new_tex->tex_data->height; j++, ly+=dly) {
  for(i=0, lx=int_to_l3d_real(0); i<new_tex->tex_data->width; i++, lx+=dlx) {

    int i_texel=(j*new_tex->tex_data->width + i) * sinfo->bytes_per_pixel;
    c2 = new_tex->tex_data->data + i_texel;

    int lumel_idx = l3d_real_to_int(ly)*polygon->lightmap->tex_data->width+l3d_real_to_int(lx);

    int lumel =  l[lumel_idx];
    sinfo->light_native(c2, lumel);
```

```
    }
  }

  return new_tex;
}

l3d_texture_data *l3d_surface_cache::
find_texdata(l3d_texture_data *texdata)
{

  int i;

  for(i=0; i<surfaces->num_items; i++) {
    if((*surfaces)[i].surface->tex_data == texdata) {

      return (*surfaces)[i].surface->tex_data;
    }
  }

  l3d_texture *new_tex;
  l3d_surface *new_surface;

  new_tex = new l3d_texture;

  int next_surface_index = surfaces->next_index();
  (*surfaces)[next_surface_index].polygon = NULL;
  (*surfaces)[next_surface_index].surface = new_tex;

  new_tex->tex_data = texdata;
  new_tex->owns_tex_data = false;

  return NULL;
}
```

Surface Cache

The surface cache is responsible for managing surfaces that have already been created. We create a surface as soon as its associated polygon is drawn for the first time. Since creating a surface is a time-consuming process (the texture data must be replicated and combined with the light map), we save already created surfaces for later reuse in the surface cache.

Class `l3d_surface_cache` is the class implementing the surface cache. Member variable `sinfo` points to the screen information object, so that we can know the pixel format of the texture data. Member variable `surfaces` is a list of pointers to surface objects. Method `reset` deletes all surfaces from the cache, thereby resetting the cache and causing all surfaces to be regenerated as they are needed. Method `find_texdata` searches the cache for a particular texture data object, returning a pointer to the data if it was found, or inserting it into the cache and returning NULL if it was not found.

Method `combine_lightmap_and_texmap` combines a light map and a texture map, returning a pointer to the new surface. This method takes a width and a height as parameters, which must be the new, possibly larger, size of the pre-tiled texture. (We'll see in the next section how we compute these size parameters.) First of all, we search to see if we have already created the surface and have stored it in the cache; if so, we return a pointer to the existing surface. If not, then we must create the surface. We do this by creating a new texture of the specified width and

height, which, remember, may be larger than the original texture size to allow for pre-tiling of the texture. We then copy the original polygon texture into the newly allocated texture, repeating the texture as necessary. Then, we loop through all texels of this pre-tiled texture, simultaneously stepping through the lumels in the light map. This is a simple scaling operation; we merely stretch the light map to the dimensions of the pre-tiled texture. Note that since the light map has a lower resolution than the texture map, several texels will map to one lumel. For each texel, we call `l3d_screen_info::light_native` with the intensity of the corresponding lumel; this changes the color of the texel to correspond to the given light level.

The question remains as to how to compute the size of the pre-tiled texture. This depends, as we said, on the relative sizes of the original texture and the polygon onto which the texture is mapped. The routine `compute_surface_orientation_and_size` in class `l3d_polygon_3d_textured_lightmapped` handles exactly this task. So, let's now take a look at the class `l3d_polygon_3d_textured_lightmapped`.

Light Mapped Polygons

Class `l3d_polygon_3d_textured_lightmapped` extends the textured polygon class to allow for light mapping.

Listing 2-21: `p3_ltex.h`

```
#ifndef __P3_LTEX_H
#define __P3_LTEX_H
#include "../../tool_os/memman.h"

#include "p3_tex.h"
#include "../surface/scache.h"

class l3d_polygon_3d_textured_lightmapped :
      virtual public l3d_polygon_3d_textured,
      virtual public l3d_texture_computer
{
  protected:
    l3d_surface_cache *scache;
  public:
    mutable l3d_texture *surface;
    //- mutable since the draw routine for s/w rendering needs to assign
    //- this member (the tiled, lit surface) if the surface is not currently
    //- in the cache, though the poly is otherwise passed as const. logically
    //- the draw routine handles the polygon as a const, physically though
    //- it can change the mutable surface member

    l3d_surface_cache *get_scache(void) const {return scache; }

    l3d_texture_space surface_orientation;
    int surface_xsize, surface_ysize;

    l3d_polygon_3d_textured_lightmapped(int num_pts, l3d_surface_cache *scache);

    ~l3d_polygon_3d_textured_lightmapped(void) {
      //- DO NOT delete the surface! it belongs to the surface cache,
      //- or might not even have been created if h/w accel with 2-pass
      //- rendering was used
```

```
      delete lightmap;
    }

    l3d_texture *lightmap;

    void compute_surface_orientation_and_size(void);

    void init_transformed(void);
    void transform(const l3d_matrix &m, int count);

    void draw(l3d_rasterizer_3d *r ) {
      r->draw_polygon_textured_lightmapped(this);
    }

    l3d_polygon_3d_textured_lightmapped
    (const l3d_polygon_3d_textured_lightmapped &r);
    l3d_polygon_2d *clone(void);

};

#endif
```

Listing 2-22: p3_ltex.cc

```
#include "p3_ltex.h"

#include "../../raster/ras3_sw.h"
#include "../../tool_os/memman.h"

#define EPSILON_VECTOR 0.001

l3d_polygon_3d_textured_lightmapped::l3d_polygon_3d_textured_lightmapped
(int num_pts, l3d_surface_cache *scache):
    l3d_polygon_2d(),
    l3d_polygon_3d(),
    l3d_polygon_3d_clippable(),
    l3d_polygon_3d_textured(num_pts)
{
  this->scache = scache;
  lightmap = new l3d_texture;
  lightmap->tex_data = new l3d_texture_data;
  lightmap->owns_tex_data = true;
  lightmap->tex_data->width = 64;
  lightmap->tex_data->height = 64;
  lightmap->tex_data->data = new unsigned char[lightmap->tex_data->width *
  lightmap->tex_data->height];

  int i,j;
  for(j=0;j<lightmap->tex_data->height;j++) {
    for(i=0;i<lightmap->tex_data->width;i++) {
      lightmap->tex_data->data[i + j*lightmap->tex_data->width] = MAX_LIGHT_LEVELS/4;
    }
  }

  surface = NULL;

}

void l3d_polygon_3d_textured_lightmapped::compute_surface_orientation_and_size
(void)
{
```

```
  O = texture->O.original;
  U = texture->U.original - O;
  V = texture->V.original - O;

  l3d_real minu,minv,maxu,maxv;

  int i= 0;

  l3d_point tex_point;
  tex_point =
    world_to_tex_matrix(*texture)
    |
    (**vlist) [(*ivertices)[i].ivertex].original;
  u = tex_point.X_ = l3d_divrr(tex_point.X_, tex_point.W_);
  v = tex_point.Y_ = l3d_divrr(tex_point.Y_, tex_point.W_);

  maxu = minu =  tex_point.X_;
  maxv = minv =  tex_point.Y_;

  U = texture->U.original - texture->O.original;
  V = texture->V.original - texture->O.original;

#define VTX(i) ((**(vlist))[ (*(clip_ivertices))[i].ivertex ].transformed)

  for(i=1; i < ivertices->num_items; i++) {

    l3d_real u,v;

    tex_point =
      world_to_tex_matrix(*texture)
      |
      (**vlist) [(*ivertices)[i].ivertex].original;
    u = tex_point.X_ = l3d_divrr(tex_point.X_, tex_point.W_);
    v = tex_point.Y_ = l3d_divrr(tex_point.Y_, tex_point.W_);

    if(u<minu) minu=u;
    if(u>maxu) maxu=u;
    if(v<minv) minv=v;
    if(v>maxv) maxv=v;
  }

  int iminu, iminv, imaxu, imaxv;

  iminu = ifloor(minu);
  iminv = ifloor(minv);
  imaxu = iceil(maxu);
  imaxv = iceil(maxv);

  int u_tilecount = imaxu - iminu;
  int v_tilecount = imaxv - iminv;

  int u_tilecount_pow2 = 1 << iceil(float_to_l3d_real(log(u_tilecount) / log(2)));
  int v_tilecount_pow2 = 1 << iceil(float_to_l3d_real(log(v_tilecount) / log(2)));
  imaxu += u_tilecount_pow2 - u_tilecount;
  imaxv += v_tilecount_pow2 - v_tilecount;

  surface_xsize = texture->tex_data->width * (imaxu - iminu);
  surface_ysize = texture->tex_data->height * (imaxv - iminv);
```

TEAMFLY

```
  surface_orientation.O.original.set(O.X_ + U.X_*iminu + V.X_*iminv,
                                     O.Y_ + U.Y_*iminu + V.Y_*iminv,
                                     O.Z_ + U.Z_*iminu + V.Z_*iminv,
                                     int_to_l3d_real(1));
  surface_orientation.U.original.set(O.X_ + U.X_*imaxu + V.X_*iminv,
                                     O.Y_ + U.Y_*imaxu + V.Y_*iminv,
                                     O.Z_ + U.Z_*imaxu + V.Z_*iminv,
                                     int_to_l3d_real(1));
  surface_orientation.V.original.set(O.X_ + U.X_*iminu + V.X_*imaxv,
                                     O.Y_ + U.Y_*iminu + V.Y_*imaxv,
                                     O.Z_ + U.Z_*iminu + V.Z_*imaxv,
                                     int_to_l3d_real(1));

  assign_tex_coords_from_tex_space(surface_orientation);
}

void l3d_polygon_3d_textured_lightmapped::init_transformed(void) {
  l3d_polygon_3d_textured::init_transformed();

  surface_orientation.O.transformed = surface_orientation.O.original;
  surface_orientation.U.transformed = surface_orientation.U.original;
  surface_orientation.V.transformed = surface_orientation.V.original;

}

void l3d_polygon_3d_textured_lightmapped::transform(const l3d_matrix &m, int count) {
  l3d_polygon_3d_textured::transform(m,count);

  surface_orientation.O.transformed = m | surface_orientation.O.transformed;
  surface_orientation.U.transformed = m | surface_orientation.U.transformed;
  surface_orientation.V.transformed = m | surface_orientation.V.transformed;

}

l3d_polygon_2d* l3d_polygon_3d_textured_lightmapped::clone(void) {
  return new l3d_polygon_3d_textured_lightmapped(*this);
}

l3d_polygon_3d_textured_lightmapped::l3d_polygon_3d_textured_lightmapped
(const l3d_polygon_3d_textured_lightmapped &r)
    : l3d_polygon_2d(r),
    l3d_polygon_3d(r),
    l3d_polygon_3d_textured(r),
    l3d_polygon_3d_clippable(r),
    l3d_texture_computer(r)
{
  scache = r.scache;
  surface = r.surface;
  surface_orientation = r.surface_orientation;
  surface_xsize = r.surface_xsize;
  surface_ysize = r.surface_ysize;
  lightmap = r.lightmap;
}
```

The member variable `scache` is a pointer to the surface cache responsible for managing the surface of the polygon. The member variable `lightmap` points to the light map, which is a small texture containing light intensities. The member variable `surface` points to the surface itself, which contains the tiled and lighted version of the texture. The surface is created by the surface cache, which is also responsible for deleting the surface.

The method `compute_surface_orientation_and_size` determines how many repetitions of the original texture appear on the polygon, based on the original texture space definition. This is done by applying the world space to texture space matrix to every vertex in the polygon, and taking the maximum and minimum resulting *u* and *v* values. Then, since a distance of 1 in texture space indicates one repetition of the texture, the number of repetitions horizontally and vertically can be computed easily as the difference between the maximum *u* or *v* value and the minimum. We round the horizontal and vertical repetition counts up to whole powers of two, then multiply them by the width and height of the original texture, respectively. We store the results in `surface_xsize` and `surface_ysize`. These variables represent the total pixel dimensions of the pre-tiled surface, which is used by the method `combine_lightmap_and_texmap`. The surface, like all textures, has dimensions that are powers of two, since this is what the original software rasterizer requires.

After computing the size of the surface, we also have to define its orientation, which we store in member variable `surface_orientation`, a texture space. We define the orientation of the surface by defining its texture space as follows. The origin of the surface's texture space is the location in the original texture space with the minimum rounded (*u*,*v*) values for the polygon. The tip of the *u* axis of the surface's texture space is the location in the original texture space with the maximum rounded *u* and minimum rounded *v* value. The tip of the *v* axis of the surface's texture space is the location in the original texture space with the minimum rounded *u* and maximum rounded *v* value. In other words, the origin of the surface's texture space lies on integer coordinates within the original texture space. The *u* axis points straight in the *u* direction in the original texture space; the *v* axis points straight in the *v* direction. Both *u* and *v* axes have tips that also lie exactly on integer coordinates in the original texture space. This special integer alignment of the surface's texture space, combined with the fact that the surface contains an integer and not a fractional number of repetitions of the texture, means that the relative positioning of the pre-tiled surface and the original texture is the same—using the pre-tiled surface and the new surface orientation appears exactly the same as using the original texture and the old texture orientation.

With the new size and orientation of the surface, we must also assign new texture coordinates to the polygon's vertices. Since the surface and its *u* and *v* axes are larger than in the original texture, the texture coordinates based on the surface will be different than (smaller than) the texture coordinates based on the original texture. We simply call `assign_tex_coords_from_tex_space` with the new surface orientation as a parameter.

The other members of class `l3d_polygon_3d_textured_lightmapped` are standard and need no lengthy explanation. The `clone` method and the copy constructor provide for duplication of the polygon through an abstract pointer. The `transform` and `init_transformed` methods have been extended to transform the light map orientation. The overridden `draw` method causes the light mapped polygon to draw itself, by asking its rasterizer to do so. Let's now look at the methods for rasterizing light mapped polygons. First, we look at software light mapping, then look at the same techniques with hardware acceleration. Surprisingly, this is one of the cases where software rasterization is simpler than hardware rasterization.

Software Rasterization of Light Maps

Because the surface cache has already pre-tiled and pre-lit the texture, the software rasterizer requires practically no changes to support light mapping. The method `draw_polygon_textured_lightmapped` in class `l3d_rasterizer_3d_sw_imp` handles rasterization of light mapped polygons.

First, we must have access to the pre-lit, pre-tiled surface. If the surface of the polygon in member variable `surface` is not currently assigned (either because the polygon is being drawn for the first time or because the surface was created but later discarded from the cache), then we create the surface by calling `combine_lightmap_and_texmap`, using the precomputed surface width and height. Then, we assign the surface orientation, stored in the polygon's variable `surface_orientation`, to the orientation stored within the surface itself. Finally, we simply assign the polygon's `texture` member variable to be the same as the new `surface` variable, and draw the polygon using the existing texture mapping routine. The surface is pre-tiled, pre-lit, and has the correct orientation. We can therefore treat it exactly like any other texture, and rasterize the polygon just like a normal texture mapped polygon.

Hardware Light Mapping

With the presence of hardware acceleration, we can use a different, more straightforward strategy for light mapping polygons. Since 3D hardware can rasterize polygons so quickly, we simply make two passes over every polygon. First, we draw the polygon textured but without lighting. Then, we draw the same polygon again, this time using the light map and not replacing the existing textured pixels, but instead altering their intensities based on the intensities stored in the light map.

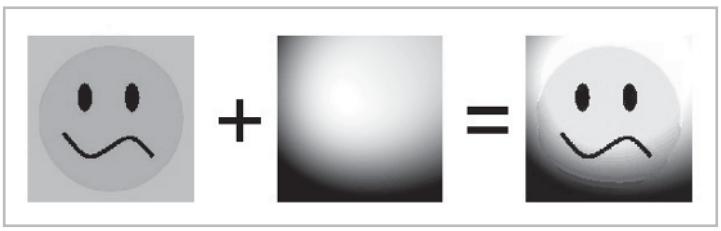


Figure 2-46: With hardware, we draw the polygon in a first pass with texture but without light map applied (left). In this first pass, the texture may be tiled. In a second pass, we apply a light map (middle) to the existing textured pixels (right). The light map is not tiled.

This two-pass approach requires three main modifications to our texture mapping strategy:

1. We don't use surfaces. In other words, we don't combine textures and light maps into one surface. Therefore, we use the surface cache to store textures and light maps separately, not surfaces.
2. We must recompute and reassign the polygon's texture coordinates to draw the texture map correctly, then we must restore the original texture coordinates, originally computed in `compute_surface_orientation_and_size`, which define the position and orientation of the light map.
3. We must tell Mesa to blend the texture map and the light map.

The Mesa rasterizer draws light mapped polygons in routine `draw_polygon_textured_lightmapped`. The routine first looks for the texture data in the cache, creating the texture if it

was not found. Texture creation takes place via the OpenGL calls we saw earlier when discussing the Mesa texture mapping routines. Next comes the tricky part: we must recompute the polygon's texture coordinates twice, once for the texture and once for the light map. The light maps have separate sizes and orientations, leading to two separate sets of texture coordinates.

To recompute the texture coordinates, we use the pixel-level approach discussed earlier. Here, we are already in the rasterization stage of the pipeline. The polygon has been projected into 2D and also clipped in 2D. Remember, the 2D clipping operation can introduce new 2D vertices which do not directly result from a 3D to 2D projection operation, meaning that for such vertices we have no 3D information. This means that we only have 2D information, but we now need to compute the texture coordinates in 3D. The Mesa rasterizer code uses the screen space to texture space formulas developed earlier. Note that it would have also been possible to use the reverse projection (covered earlier in the discussion of the Mesa texture mapping routines) to retrieve the 3D world coordinates from the pixel coordinates, then use the world-to-texture matrix to find the texture coordinates, but it is easier to use the pixel-level formulas, since they go directly from 2D screen coordinates to texture coordinates.

Let's step through the new computation of the texture coordinates for the texture map.

First, we start a loop for all post-projection, clipped 2D vertices:

```
for(int i=0; i<p_poly->clip_ivertices->num_items; i++) {
```

Then, we compute the simple projection coordinates from the actual screen coordinates. This undoes the effect of the field of view correction, the `y` axis reversal, and the top-left instead of screen-centered origin. We do this computation because the simple projection coordinates are what we need for the screen space to texture space formulas. Notice that to perform this computation, we need the field of view terms and the screen size, which is why these variables are referenced in the `l3d_texture_computer` class.

```
float px =
  ( (**p_poly->vlist) [(*p_poly->clip_ivertices)[i].ivertex].transformed.X_
    - *screen_xsize/2 )
  / (*screen_xsize * *fovx);

float py =
  (*screen_ysize/2
   -
   (**p_poly->vlist) [(*p_poly->clip_ivertices)[i].ivertex].transformed.Y_ )
  /
  (*screen_ysize * *fovy);
```

Next, given the simple projection coordinates, we simply implement the screen space to texture space conversion formulas derived earlier. We compute *u*, *v*, and *z* coordinates for the current vertex—in other words, the horizontal and vertical positions in the texture space of the texture map, and the camera space *z* coordinate.

```
float ox = p_poly->texture->O.transformed.X_;
float oy = p_poly->texture->O.transformed.Y_;
float oz = p_poly->texture->O.transformed.Z_;
float ux = p_poly->texture->U.transformed.X_ - ox;
float uy = p_poly->texture->U.transformed.Y_ - oy;
float uz = p_poly->texture->U.transformed.Z_ - oz;
float vx = p_poly->texture->V.transformed.X_ - ox;
float vy = p_poly->texture->V.transformed.Y_ - oy;
float vz = p_poly->texture->V.transformed.Z_ - oz;
```

```
float u = (px*(oy*vz - oz*vy) + py*(oz*vx - ox*vz) - oy*vx + ox*vy)
          / (px*(uz*vy - uy*vz) + py*(ux*vz - uz*vx) + uy*vx - ux*vy);
float v = (px*(oy*uz - oz*uy) + py*(oz*ux - ox*uz) - oy*ux + ox*uy)
          / ((uy*vz - uz*vy)*px + py*(uz*vx - ux*vz) + ux*vy - uy*vx);
float z = (ox*(uz*vy - uy*vz) + oy*(ux*vz - uz*vx) + oz*(uy*vx - ux*vy))
          / (px*(uz*vy - uy*vz) + py*(ux*vz - uz*vx) + uy*vx - ux*vy);
```

The last step is to store these values with the vertex (more precisely, the vertex index). We also save the original texture coordinates, defining the light map orientation, so we can restore them later when we are drawing the light map.

```
old_texcoords[old_texcoords.next_index()] =
  ((l3d_polygon_ivertex_textured *)&(*p_poly->clip_ivertices)[i])
  ->tex_coord;

((l3d_polygon_ivertex_textured *)&(*p_poly->clip_ivertices)[i])
->tex_coord.set(float_to_l3d_real(u),
                float_to_l3d_real(v),
                float_to_l3d_real(z),
                float_to_l3d_real(1.0));
```

Once we have computed the *u*, *v*, and *z* values, based on the texture map orientation, for all vertex indices within the polygon, we then draw the polygon using the previous Mesa texture mapping routine, `draw_polygon_textured`.

At this point, we now have the texture mapped polygon drawn. Next, we make a second pass to add in the effect of the light map. Before we make the second pass, though, we must tell OpenGL that we want the next texture that is drawn, which will be the light map, to not overwrite the current pixels, but instead to alter their intensity. The following lines do this:

```
//- set up mag/min filters to make light maps smoother
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

//- set up blend function to combine light map and text map
glEnable(GL_BLEND);
glBlendFunc(GL_ZERO, GL_SRC_ALPHA);
```

The two calls to `glTexParameteri` tell OpenGL to smooth the light maps. A texture map or light map is (practically) always displayed on-screen either larger or smaller than its original dimensions; in other words, it is very rare that one pixel in the texture map maps to exactly one and only one pixel in the final image. More often, a texel in the texture map maps to either less than one or more than one pixel in the final image. This leads to either stretching of the texture image, which OpenGL calls *maxification*, or squashing of the image, which OpenGL calls *minification*. In such cases, we can either choose the nearest texel to display, which is what we have been doing so far in both the software and Mesa rasterizers, or we can compute an average of the nearest texel and the surrounding texels, displaying the averaged color. Doing such averaging is, of course, slower than simply choosing the nearest texel, but it also generally results in a better quality image, smoothing out the changes in a texture map and making the individual texels harder to discern. Since light maps are typically low resolution, and since we are using Mesa mainly as an interface to graphics acceleration hardware, we go ahead and turn on the averaging, assuming that the underlying graphics acceleration hardware is capable of handling the additional computations. The parameter GL_LINEAR tells OpenGL to perform a weighted linear average on the texels.

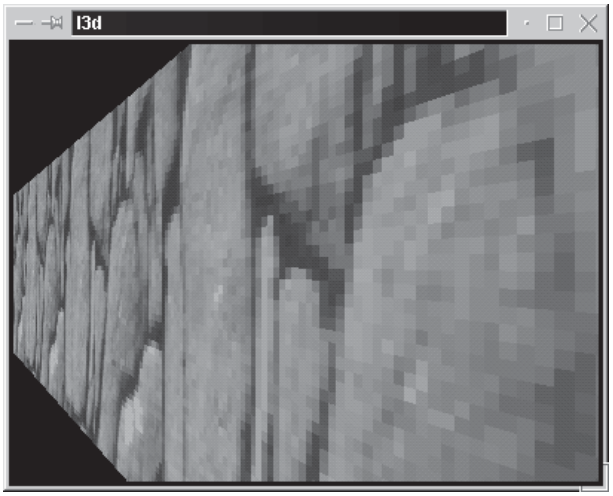


Figure 2-47: Normal texture mapping without linear filtering applied. Notice that the individual texels of the texture image are clearly visible.

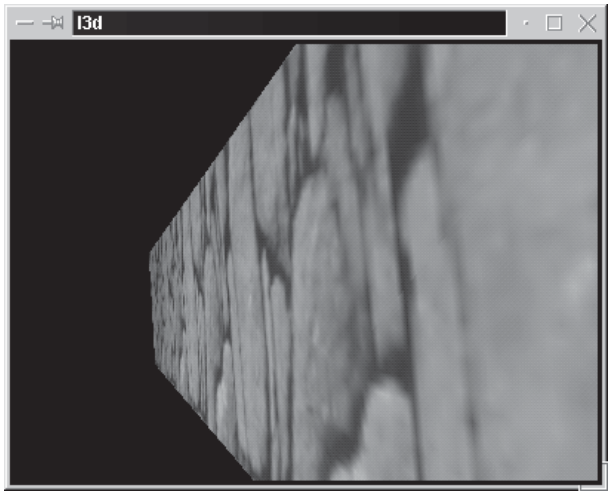


Figure 2-48: Texture mapping with linear filtering applied. Notice that the individual texels are no longer visible, but that the image is slightly blurrier as a result.

Then, we also need to tell OpenGL to blend the light map with the existing pixels from the texture map which were already drawn in the first pass. To do this, we call `glEnable(GL_BLEND)`, which enables some sort of blending for the coming drawing operations, and then call `glBlendFunc(GL_ZERO, GL_SRC_ALPHA)` to specify exactly how the already existing pixels should be blended with the new pixels to be drawn from the light map. The two parameters to `glBlendFunc` control the relative weight of the new (or source) and the existing (or destination) pixels, respectively. Thus, the first parameter `GL_ZERO` means that we display none of the color of the light map. The second parameter `GL_SRC_ALPHA` means that we display the existing color from the texture map, but using the value from the source pixel (the light map) as an alpha value. In computer graphics, an alpha value is simply a transparency value; the greater the alpha, the more transparent a pixel is. Thus, by treating the light map values as transparency values, we can "see through" the brighter areas of the light map to see more of the underlying light from the

texture; in darker areas of the light map, we see less or none of the original texture. This is exactly the desired illumination effect.

Then, the rest of the routine `draw_polygon_texture_lightmapped` draws the same polygon again, only this time with the light map as the texture. First, we look for the light map data in the surface cache, creating it if necessary with OpenGL calls. Then, we restore the original light map texture coordinates for each vertex, which we saved during the first texture coordinate recomputation. Finally, we draw the polygon just as we did in the `draw_polygon_textured` routine, performing a reverse projection using the polygon's plane to obtain the pre-projection coordinates Mesa needs to do perspective correct texture mapping.

Sample Program: lightmap

The last sample program in this chapter illustrates the use of light mapping and texture mapping combined. We also see an implementation of the actual computation of light intensities for the light maps, by using the diffuse reflection equations discussed earlier.

The program `lightmap` displays a number of static pyramid objects, all initially dark. You can fly around the scene with the standard navigation keys provided by class `l3d_pipeline_world`. Press **p** to place a point light at the current camera position. This causes a computation of light intensity for every lumel of every light map in the scene. Then, we reload all light maps by resetting the surface cache. We, therefore, see the effect of the light immediately. You can place any number of lights in the scene. The lights themselves are not stored in the program; only their effect on the light maps is stored.

The program actually displays light mapping and texture mapping simultaneously. The effect of the light mapping is easier to see with a blank texture, which is the default. To see light mapping with a real texture, type **cp test.ppm.colored test.ppm** when in the program directory. This copies a colored image into the file `test.ppm`, which is the texture file used by the program. Then run the program as usual. To return to the blank texture image, type **cp test.ppm.white test.ppm**.

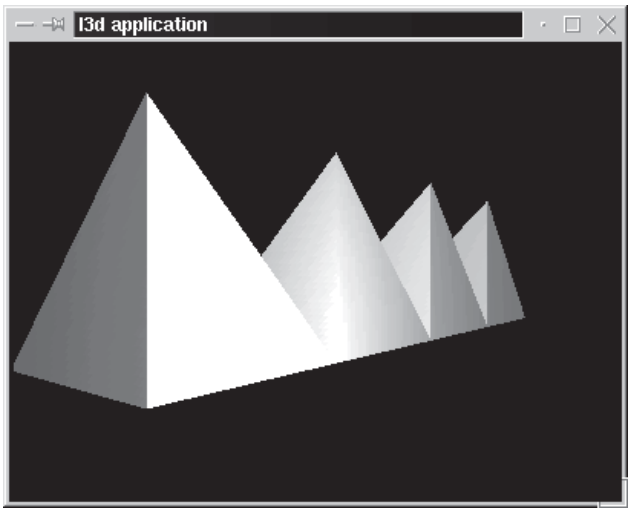


Figure 2-49: Output from the sample program `lightmap`.

Listing 2-23: shapes.h, the declaration of the light mapped pyramid objects for sample program lightmap

```
#include "../lib/geom/object/object3d.h"
#include "../lib/geom/polygon/p3_ltex.h"
#include "../lib/geom/texture/texture.h"
#include "../lib/geom/texture/texload.h"

class pyramid:public l3d_object {
  public:
    pyramid(l3d_texture_loader *l, l3d_surface_cache *scache);
    virtual ~pyramid(void);
};
```

Listing 2-24: shapes.cc, the implementation of the light mapped pyramid objects for sample program lightmap

```
#include "shapes.h"

#include <stdlib.h>
#include <string.h>

pyramid::pyramid(l3d_texture_loader *l, l3d_surface_cache *scache) :
    l3d_object(4)
{
  (*vertices)[0].original.set(float_to_l3d_real(0.),float_to_l3d_real(0.),float_to_l3d_real(0.),
          float_to_l3d_real(1.));
  (*vertices)[1].original.set(float_to_l3d_real(50.0),float_to_l3d_real(0.),float_to_l3d_real(0.),
          float_to_l3d_real(1.));
  (*vertices)[2].original.set(float_to_l3d_real(0.),float_to_l3d_real(50.),float_to_l3d_real(0.),
          float_to_l3d_real(1.));
  (*vertices)[3].original.set(float_to_l3d_real(0.),float_to_l3d_real(0.),float_to_l3d_real(50.),
          float_to_l3d_real(1.));

  l3d_polygon_3d_textured_lightmapped *p;
  int pi;

  pi = polygons.next_index();
  polygons[pi] = p = new l3d_polygon_3d_textured_lightmapped(3,scache);
  polygons[pi]->vlist = &vertices;
  (*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=0;
  (*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=1;
  (*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=3;
  l->load("test.ppm");
  p->texture = new l3d_texture;
  p->texture->tex_data = new l3d_texture_data;
  p->texture->tex_data->width = l->width;
  p->texture->tex_data->height = l->height;
  p->texture->tex_data->data = l->data;
  p->texture->owns_tex_data = true;
  p->texture->O = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[0].ivertex];
  p->texture->U = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[1].ivertex];
  p->texture->V = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[2].ivertex];
  polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();
  p->assign_tex_coords_from_tex_space(*p->texture);
  p->compute_surface_orientation_and_size();
  p->plane.align_with_point_normal(p->center.original, normalized(p->sfcnormal.original -
          p->center.original));

  pi = polygons.next_index();
  polygons[pi] = p = new l3d_polygon_3d_textured_lightmapped(3,scache);
```

```
polygons[pi]->vlist = &vertices;

(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=2;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=3;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=1;

p->texture = new l3d_texture;
p->texture->tex_data = new l3d_texture_data;
p->texture->tex_data->width = l->width;
p->texture->tex_data->height = l->height;
p->texture->tex_data->data = l->data;
p->texture->owns_tex_data = true;

p->texture->O = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[0].ivertex];
p->texture->U = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[1].ivertex];
p->texture->V = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[2].ivertex];

polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();
p->assign_tex_coords_from_tex_space(*p->texture);
p->compute_surface_orientation_and_size();
p->plane.align_with_point_normal(p->center.original, normalized(p->sfcnormal.original -
        p->center.original));

pi = polygons.next_index();
polygons[pi] = p = new l3d_polygon_3d_textured_lightmapped(3,scache);
polygons[pi]->vlist = &vertices;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=0;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=2;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=1;

p->texture = new l3d_texture;
p->texture->tex_data = new l3d_texture_data;
p->texture->tex_data->width = l->width;
p->texture->tex_data->height = l->height;
p->texture->tex_data->data = l->data;
p->texture->owns_tex_data = true;

p->texture->O = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[0].ivertex];
p->texture->U = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[1].ivertex];
p->texture->V = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[2].ivertex];

polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();
p->assign_tex_coords_from_tex_space(*p->texture);
p->compute_surface_orientation_and_size();
p->plane.align_with_point_normal(p->center.original, normalized(p->sfcnormal.original -
        p->center.original));

pi = polygons.next_index();
polygons[pi] = p = new l3d_polygon_3d_textured_lightmapped(3,scache);
polygons[pi]->vlist = &vertices;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=3;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=2;
(*(polygons[pi]->ivertices))[polygons[pi]->ivertices->next_index()].ivertex=0;

p->texture = new l3d_texture;
p->texture->tex_data = new l3d_texture_data;
p->texture->tex_data->width = l->width;
p->texture->tex_data->height = l->height;
p->texture->tex_data->data = l->data;
```

```
  p->texture->owns_tex_data = true;

  p->texture->O = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[0].ivertex];
  p->texture->U = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[1].ivertex];
  p->texture->V = (**(polygons[pi]->vlist))[(*(polygons[pi]->ivertices))[2].ivertex];

  polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();
  p->assign_tex_coords_from_tex_space(*p->texture);
  p->compute_surface_orientation_and_size();
  p->plane.align_with_point_normal(p->center.original, normalized(p->sfcnormal.original -
          p->center.original));

  num_xforms = 1;
  modeling_xforms[0].set
  ( float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.),
    float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.),
    float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.),
    float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.) );

  modeling_xform=
    modeling_xforms[0];
}

pyramid::~pyramid(void) {
  for(register int i=0; i<polygons.num_items; i++) {delete polygons[i]; }
}
```

Listing 2-25: `main.cc`, the main program file for sample program `lightmap`

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../lib/geom/object/object3d.h"
#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rasteriz.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/geom/world/world.h"
#include "../lib/system/fact0_2.h"
#include "../lib/geom/texture/texture.h"
#include "../lib/geom/texture/tl_ppm.h"
#include "../lib/pipeline/pi_wor.h"

#include "shapes.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdarg.h>

class my_pipeline : public l3d_pipeline_world {
  public:
    my_pipeline(l3d_world *w) :
    l3d_pipeline_world(w) {}
    void key_event(int ch);
};

class my_world :
```

TEAM-FLY

```
      public l3d_world,
      public l3d_texture_computer
{
  public:
    my_world(void);
    void update_all(void);

    virtual ~my_world(void) {
      delete texture_loader;
      delete scache;
    }
    l3d_texture_loader *texture_loader;
    l3d_surface_cache *scache;

    int compute_light_level(l3d_real u, l3d_real v);
    void place_lamp(void);
    void reset_surface_cache(void) {
      scache->reset();
    }
};

void my_pipeline::key_event(int c) {
  l3d_pipeline_world::key_event(c);

  switch(c) {
    case 'p': {
        my_world *w;
        w = dynamic_cast<my_world *>(world);
        if(w) {
          w->place_lamp();
          w->reset_surface_cache();
        }
      }
  }
}

main() {
  l3d_dispatcher *d;
  my_pipeline *p;
  my_world *w;

  factory_manager_v_0_2.choose_factories();
  d = factory_manager_v_0_2.dispatcher_factory->create();

  w = new my_world();
  p  = new my_pipeline(w);
  d->pipeline = p;
  d->event_source = w->screen;

  d->start();

  delete p;
  delete d;
  delete w;
}

void my_world::update_all(void) {
  l3d_world::update_all();
}
```

```
my_world::my_world(void)
    : l3d_world(400,300)
{

  rasterizer_3d_imp->fovx = &(camera->fovx);
  rasterizer_3d_imp->fovy = &(camera->fovy);
  rasterizer_3d_imp->screen_xsize = &(screen->xsize);
  rasterizer_3d_imp->screen_ysize = &(screen->ysize);

  l3d_screen_info *si = screen->sinfo;
  texture_loader = new l3d_texture_loader_ppm(screen->sinfo);

  camera->VRP.set(0,0,-50,0);
  camera->near_z = float_to_l3d_real(5.5);
  camera->far_z = int_to_l3d_real(500);

  scache = new l3d_surface_cache(screen->sinfo);

  int i,j,k,onum;

  i=10; j=0; k=20;
  k=0;

  for(i=1; i<200; i+=50) {
    objects[onum = objects.next_index()] = new pyramid(texture_loader,scache);

    if (objects[onum]==NULL) exit;
    objects[onum]->modeling_xforms[0].set
    ( int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(i),
      int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(1),
      int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(1),
      int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1) );
    objects[onum]->modeling_xform =
      objects[onum]->modeling_xforms[0] ;
  }

  screen->refresh_palette();
}

int my_world::compute_light_level
(l3d_real u, l3d_real v)
{
  l3d_real x,y,z;

  x = O.X_ + l3d_mulri(U.X_, u) + l3d_mulri(V.X_, v);
  y = O.Y_ + l3d_mulri(U.Y_, u) + l3d_mulri(V.Y_, v);
  z = O.Z_ + l3d_mulri(U.Z_, u) + l3d_mulri(V.Z_, v);

  l3d_point aLight(0,0,0,1);

  l3d_vector L(aLight.X_ - x ,
               aLight.Y_ - y ,
               aLight.Z_ - z ,
               int_to_l3d_real(0)),

  N(cross(U,V));
  l3d_real intensity;

  l3d_real f_att=l3d_divrr(int_to_l3d_real(1) ,
  (float_to_l3d_real(0.1)+float_to_l3d_real(0.02*sqrt(l3d_real_to_float(dot(L,L))))));
```

```
          if (f_att>int_to_l3d_real(1)) f_att=int_to_l3d_real(1);

  intensity = l3d_mulrr ( l3d_mulrr(int_to_l3d_real(MAX_LIGHT_LEVELS/2) , f_att) ,
                          dot(normalized(N),normalized(L)) );
  if (intensity>int_to_l3d_real(MAX_LIGHT_LEVELS/2)) {intensity = int_to_l3d_real(MAX_LIGHT_LEVELS/2);
}

  if (intensity<int_to_l3d_real(0))  {intensity = int_to_l3d_real(0);  }

  return l3d_real_to_int(intensity);
}

void my_world::place_lamp(void)
{
  int iObj, iFacet;

  l3d_vector facet_to_cam_vec;
  l3d_vector N;

  for(iObj=0; iObj<objects.num_items; iObj++) {
    l3d_object *object = objects[iObj];

    l3d_polygon_3d_clippable *pc;

    object->init_nonculled_list();
    object->vertices->num_varying_items = 0;
    object->transform_stage = 0;
    for(register int ivtx=0; ivtx<object->vertices->num_fixed_items; ivtx++) {
      (*(object->vertices))[ivtx].reset();
    }

    l3d_polygon_3d_node *n;
    n = object->nonculled_polygon_nodes;
    while(n) {
      n->polygon->init_clip_ivertices();
      n->polygon->init_transformed();
      n=n->next;
    }

    if (object->num_xforms) {
      object->transform(object->modeling_xform);
    }

    object->camera_transform(camera->viewing_xform);

    n = object->nonculled_polygon_nodes;
    while(n) {
      l3d_polygon_3d_textured_lightmapped *pt;

      pt = dynamic_cast<l3d_polygon_3d_textured_lightmapped *>
           (n->polygon);
      if(pt) {
        int lumel_y,lumel_x;

        O = pt->surface_orientation.O.transformed;
        U = pt->surface_orientation.U.transformed - O;
        V = pt->surface_orientation.V.transformed - O;
        printf("obj %d ", iObj);
        O.print();
        U.print();
```

```
        V.print();

        for(lumel_y=0; lumel_y < pt->lightmap->tex_data->height; lumel_y++) {
          for(lumel_x=0; lumel_x<pt->lightmap->tex_data->width;lumel_x++) {
            l3d_real u,v;
            u = l3d_divri(int_to_l3d_real(lumel_x),
                          pt->lightmap->tex_data->width);
            v = l3d_divri(int_to_l3d_real(lumel_y),
                          pt->lightmap->tex_data->height);
            int new_light =
              (int)(pt->lightmap->tex_data->data
                    [ lumel_y * pt->lightmap->tex_data->width + lumel_x ]) +
              compute_light_level(u, v);

            if ( new_light > MAX_LIGHT_LEVELS ) new_light = MAX_LIGHT_LEVELS;

            pt->lightmap->tex_data->data
            [ lumel_y * pt->lightmap->tex_data->width + lumel_x ] = new_light;

          }
        }
      }

      n = n->next;
    }
  }
}
```

The `pyramid` class is the same as in the previous program `textest`; the only differences are the pyramid does not rotate and the polygons used to make up the pyramid are of type `l3d_polygon_3d_textured_lightmapped`. Since all important functions on polygons are virtual functions, we need only create instances of our new light mapped polygons and assign them to our 3D objects. Notice that the constructor of the pyramid, as well as the constructor of the light mapped polygons, require a surface cache as a parameter. The surface cache is created and destroyed by our custom world class, `my_world`.

The custom world class offers two new methods for light mapping: `place_lamp` and `compute_light_level`. Method `place_lamp` transforms all polygons in the scene into camera space. Since all polygons are light mapped polygons, this means that all light map orientations are also transformed into camera space. Then, for each lumel in each light map, we compute the light level by calling `compute_light_level`. This illuminates all lumels of all polygons in the scene.

Method `compute_light_level` first converts the (u,v) coordinates of the lumel back into camera space, by using the light map's orientation vectors (as a coordinate system), which have also been translated into camera space. We then compute the light intensity of the lumel using the diffuse lighting equation presented earlier. The light position is set to be at the origin—in other words, the location of the camera. We clamp the maximum light intensity from the lighting equation only to be half of the maximum allowable light intensity. This ensures that cumulative applications of light can still cause an increase in intensity.



NOTE We convert from 2D lumel (*u*,*v*) space into 3D camera space (3D space with the camera at the origin in the standard orientation), instead of converting from 2D lumel space into 3D world space (3D space with the camera somewhere other than the origin and oriented arbitrarily), then perform the lighting calculations in 3D camera space. It would also be possible to convert from lumel space to world space, by applying the inverse of the camera transformation matrix to go from lumel space to camera space to world space. Then, we would perform the lighting computations in world space. But this extra conversion to world space is unnecessary. We just need to convert from 2D lumel space into some 3D space where we can determine the 3D light positions, 3D light vectors, and 3D polygon surface normal vectors, so that we can perform the 3D lighting computations. Both camera space and world space are fine for this purpose. In this particular example, it is simply easier to use camera space coordinates, though you may find it more intuitive to think of lighting computations as occurring in world space. As long as we can examine the relative 3D positions of lights, light vectors, and normal vectors, it doesn't matter which 3D coordinate system we use to do the lighting computations.

Shadows and Light Maps

The lighting computation in the last sample program does not take into account the fact that some objects obstruct other objects. Light does not travel through opaque objects, creating areas of shadow on objects behind the obstructing object. Shadow computation can be done by performing an additional test before lighting each lumel. We check to see if any polygon from any object lies between the light source and the camera space (or world space) coordinates of the lumel. If there is such a polygon, the light rays from the light source are blocked and do not reach the lumel in question; therefore, we proceed computing the light for the next lumel. If there is no such polygon, the light rays from the light source reach the target lumel, and the lighting can be applied as normal.

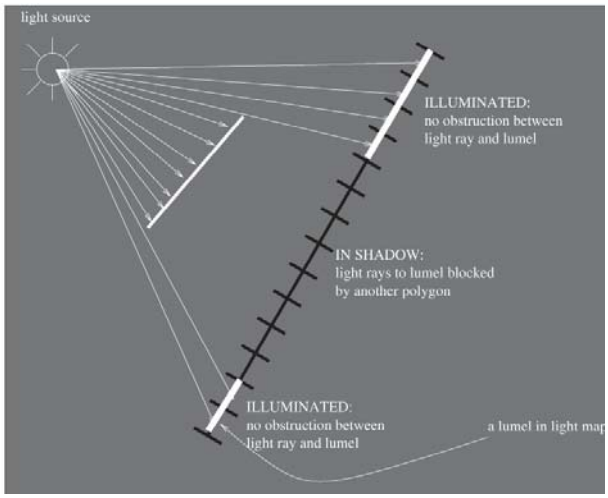


Figure 2-50: Shadow computation with light maps.

Testing to see whether any polygon lies between the light source and the lumel requires the creation of a vector from the light source to the lumel and an intersection test to see if the vector intersects any polygon in the scene. Creating the vector is easy: its starting point in 3D is the light

source, and its ending point in 3D is the lumel. The intersection test is harder. We learned how to intersect a parametric line with a plane when we looked at clipping against arbitrarily oriented planes. The problem here is that we don't want to intersect a line with an entire infinite plane, but only with a polygon—a tiny polygonal cut-out portion of a plane. This requires the line-plane test developed earlier in this chapter, plus an additional test to see if the point of intersection lies within the bounds of the polygon. Chapter 8 develops exactly this line-polygon intersection test, for the purposes of collision detection.

With the line-polygon intersection test defined, computing shadows simply reduces to a brute-force check to see if any polygon obstructs any light source for every lumel in the scene. This works, but requires a lot of computation. One way of getting around this is to divide the scene up into separate larger regions, such as cubes, with each region containing many polygons. We then intersect the light ray with the boundaries of region. If the light ray does not intersect a region, then it cannot intersect any of the polygons in the region. This can reduce the total amount of work needed to compute shadows. A related way of computing shadows has to do with an explicit visibility encoding by means of portals (see Chapter 5).

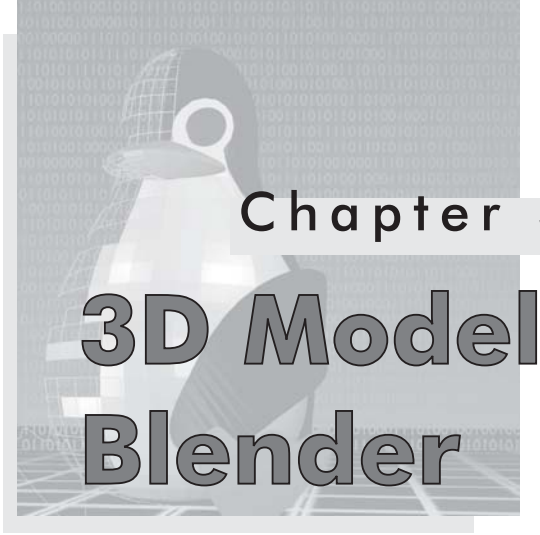


NOTE There are also many other ways of computing shadows. Visible surface determination, covered in Chapters 4 and 5, is a huge area in 3D graphics. Computing shadows is also essentially a visible surface problem; we must determine which surfaces are visible from the viewpoint of the light source. Surfaces which can be "seen" by the light are illuminated; those invisible to the light are in shadow.

Summary

In this chapter, we saw several visual techniques for 3D polygons. These techniques include 3D morphing, light computation, texture mapping with PPM image files, and light mapping. We solved some equations using the symbolic algebra package Calc, and observed the linearity of u/z, v/z, and $1/z$ in screen coordinates, and the non-linearity of u, v, and z in screen coordinates.

We now have a pretty good handle on 3D polygons, but our sample programs have still used fairly basic 3D objects: pyramids consisting of just a few triangles. Creating more complicated objects requires the use of a 3D modeling package. The next chapter covers using the free 3D modeling package Blender to create more interesting models for use in our 3D programs.



Chapter 3

3D Modeling with Blender

The Blender 3D modeling package is a powerful tool for creating 3D objects. Chapter 1 already reviewed the basics of using Blender to create simple polygonal models for import into an l3d program. In this chapter, we create more complex models, illustrating the use of Blender's animation and texture mapping features. In particular, we cover the following topics:

- Assigning texture coordinates to models
- Using VRML to export and import texture coordinates
- Creating compatible morph targets
- Viewing animations in Blender
- Using inverse kinematics and rotoscoping

This chapter is organized around two tutorials. The first illustrates a 3D morph between a cube and a texture-mapped tree. The second creates a jointed, animated human figure.

Tutorial: Creating and Exporting Compatible, Textured 3D Morph Targets

We already know how to import arbitrary models into our 3D programs by using our Videoscape plug-in for l3d objects. This tutorial takes the idea a step further, illustrating how to use Blender to apply textures to our models, and how to import these applied textures into our 3D programs. Furthermore, this tutorial also illustrates how to create *compatible morph targets*—two models with the same number and ordering of vertices and faces that can morph into one another. Chapter 2 illustrated morphing in 3D, but with simple, uninteresting objects. Now it's time to create an interesting model in Blender and morph it.

Let's start with the morphing aspect; we then deal with applying texture coordinates after we have generated the morph targets.

First, clear everything in Blender by pressing **Ctrl+x**. Delete the default plane object.

The Starting Morph Mesh

As we touched upon in Chapter 2, in order for two models to be able to morph into one another, they must have the same number and ordering of vertices and faces. The reason for this is that morphing is simply interpolation of vertex positions from old positions to new positions, which is only (easily) possible if the topology of both models is the same. The best way to ensure this is to create both source and target models by using the same basis mesh in Blender, merely deforming the mesh within Blender to shape it into the source and target models. It is not permitted to add or delete vertices or faces; only deformation of the mesh (i.e., displacement of the mesh's vertices) is allowed.

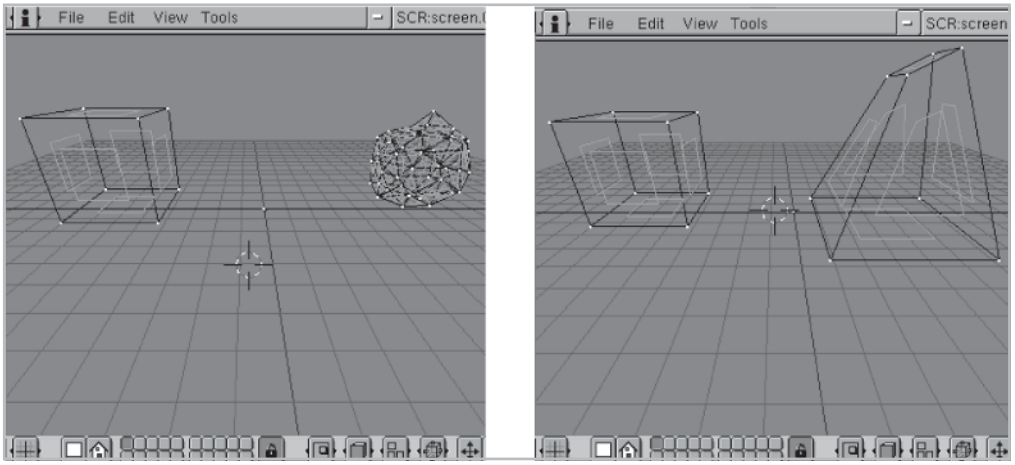


Figure 3-1: Incompatible (left) and compatible (right) morph targets.

We'll start out with a cube.

1. Move the 3D cursor to the origin if it is not already there.
2. Add a cube. Remain in EditMode, and make sure that all vertices are selected (displayed in yellow).
3. Type **w**. The Specials pop-up menu appears. Select **Subdivide**. Blender subdivides every selected face into four smaller, equally sized faces. Subdividing the cube in this manner preserves the shape of the cube, but gives us more faces to work with.
4. Subdivide the cube again. Each side of the cube should now have 16 faces (4 4).
5. Scale the cube up, so that each side is approximately 8 grid units long.

Now, we're going to define two morph targets for this mesh, then edit these targets.

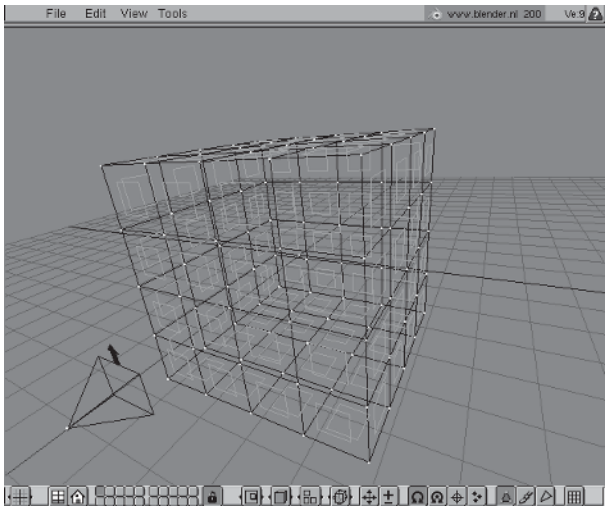


Figure 3-2

Inserting Two Morph Targets into Blender

The next step is to insert two morph targets into Blender, so that we can see and test the morph within Blender itself. Blender refers to these morph targets as *vertex keys*, a term which comes from key-frame animation. Insert two vertex keys as follows:

1. Ensure that Blender is in EditMode.
2. Notice the numerical slider button on the right-hand side of the header of the ButtonsWindow. This slider indicates the current frame number for animations. Ensure that the value of this slider is 1. If not, change its value by dragging it with the mouse or by pressing **Down Arrow** (decrease frame by 10) or **Left Arrow** (decrease frame by 1) until its value is 1.
3. Type **i** to insert a key. A pop-up menu appears, allowing you to select different kinds of keys. Select **Mesh**. This saves the current positions of the vertices of the mesh into a key.

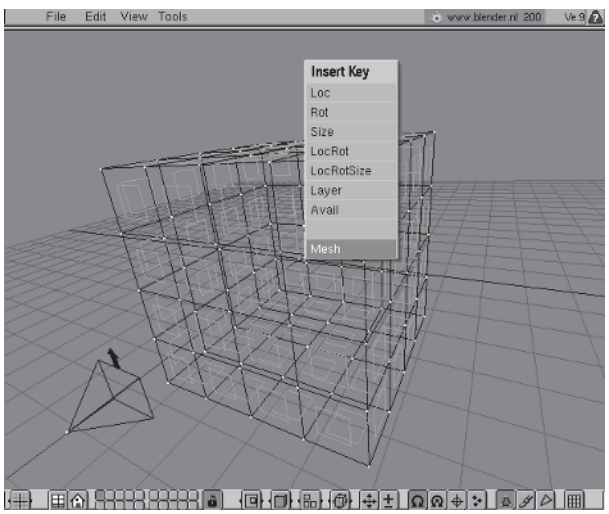


Figure 3-3

4. Exit and re-enter EditMode by pressing **Tab** twice.
5. Increase the frame number to 21 by pressing **Up Arrow** (increase frame by 10) twice, or press **Right Arrow** (increase frame by 1) 20 times.
6. Type **i** to insert another key.

We now have inserted two vertex keys. You can see these keys by looking in an IpoWindow as follows.

1. Exit EditMode. Select the cube object for which you just inserted the vertex keys.
2. Open an IpoWindow, preferably in a new window next to the 3DWindow (as opposed to changing the existing 3DWindow to be of type IpoWindow).
3. In the window header of the IpoWindow, click on the ICONBUT with the image of a polygon with vertices. This displays vertex keys in the IpoWindow.
4. Ensure that the ICONBUT with the image of a key is not active.

Notice the two horizontal lines within the window. The bottommost orange line represents the first, baseline vertex key. The first vertex key is always displayed in orange. The second and all additional vertex keys are displayed as blue horizontal lines. The order of the vertex keys is from bottom to top.

Also notice the curved line. This is the interpolation curve, which interpolates from the first key to the second key (vertically) over time (horizontally). The vertical green line indicates the current frame number. If you change the current frame with the arrow keys, the green line moves left or right along the time axis accordingly.

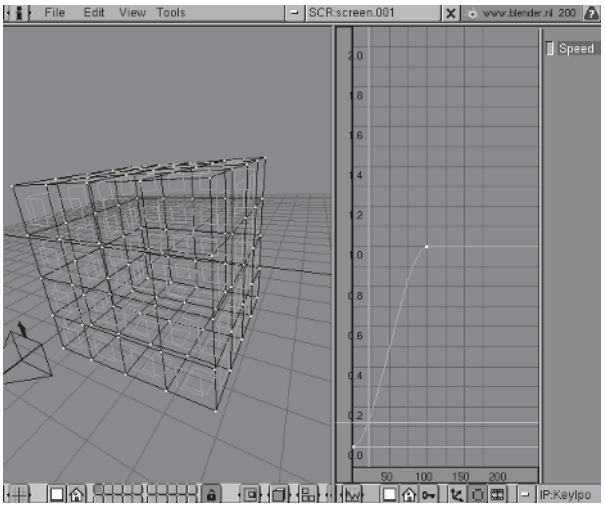


Figure 3-4: The IpoWindow with the two vertex keys.

Here are some operations you can perform in the IpoWindow. Don't try these out just now because the current configuration is important for completing this tutorial. You can right-click a vertex key in the IpoWindow to select it, at which point it changes to a lighter color (light orange for the first vertex key, light blue for the additional vertex keys). Type **g** to grab and move the selected vertex key(s) to another location, or **x** to delete a vertex key. If you add vertex keys in the

Team-Fly®

3DWindow (remember that you must change the frame number to insert multiple vertex keys), you can see how more and more blue lines (the vertex keys) appear within the IpoWindow.

For now, make sure that only two vertex keys are present, and that the second vertex key, the topmost one, is selected (displayed in light blue).

Deforming the Mesh

We now have two vertex keys defined. The second one is selected, because it was the last one inserted (or because you right-clicked it in the IpoWindow).

With a vertex key selected, any changes to the mesh apply to that vertex key only. This means that if we go into EditMode now and change the mesh, it changes the mesh just for the second vertex key, but the first vertex key still retains the original shape of the mesh.

This means that we now need to enter EditMode and deform the mesh into a new shape. Begin as follows.

1. Select the upper-right vertex of the cube.
2. Move the vertex somewhere away from the corner, so that a sharp point is created.
3. Exit EditMode. At this point, by exiting EditMode with a modified mesh, we have just saved the current shape into the currently selected vertex key.

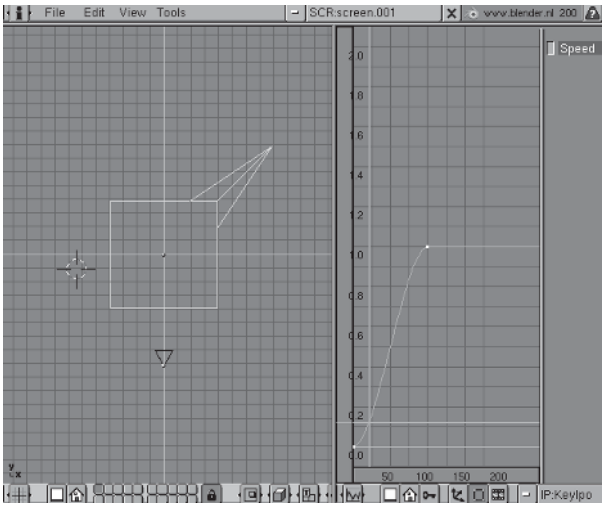


Figure 3-5

4. Press **Left Arrow** a few times to decrease the frame number by a few frames. Notice that the sharp point you created at the corner of the mesh slowly starts to disappear. As you decrease the frame number, Blender automatically interpolates the vertex positions to the appropriate vertex key (computed by Blender by intersecting the interpolation curve in the IpoWindow with the vertex key for the current frame number), which in this case is the first vertex key. Since the first vertex key contains the unmodified cube, the mesh morphs back to the original cube the closer the frame number is to the first frame.

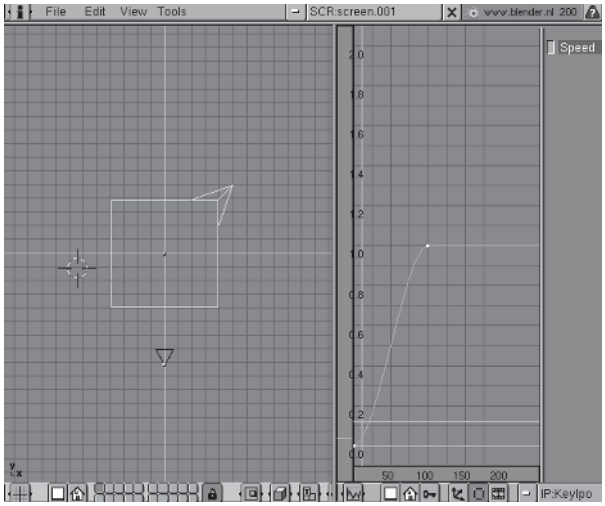


Figure 3-6

5. Press **Right Arrow** to return the frame number back to frame 21. Notice that the sharp point reappears as you approach frame 21, the time at which the second vertex key was inserted.

Importantly, and quite fortunately, if we save a Videoscape mesh from Blender, the exported geometry reflects the current state of the mesh for the currently active frame. This means that to export two compatible morph targets, we simply change to frame 1, write the first mesh, then change to frame 21, at which point Blender automatically interpolates the vertices to the second vertex key, and write a second mesh.

Now, you should again be at frame 21. Your job now is to enter EditMode and deform the mesh into something interesting. Any shape will do, but for this tutorial I chose to deform the mesh into the shape of a palm tree. Here are some hints for making the palm tree:

1. Start by making the bottom half of the cube longer and thinner to represent the trunk. Next, make the top flat and somewhat x-shaped. Finally, pull down the outer edges of the top x to make the leaves droop down.
2. Select groups of vertices using border select mode, and translate, rotate, or scale these groups of vertices. Creating the tree by moving each vertex individually would be very time consuming.
3. Be careful not to pull vertices around in physically impossible ways. For instance, if a vertex is originally on one side of an edge in the original model, you should not pull the vertex such that it is on the other side of the edge. If you do so, this would reverse the front/back orientation of the faces connected with the vertex, which leads to incorrect display later. Think of yourself as a sculptor working with a malleable balloon; you can stretch or compress the balloon to change its shape, but if you try to poke the balloon through itself, the whole thing blows up in your face.

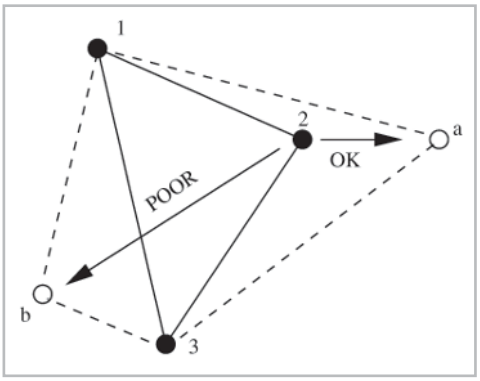


Figure 3-7: Avoid pulling vertices across edges where the orientation would change. In this example, the vertex 2 may be pulled to the point marked a while retaining the same orientation. On the other hand, pulling vertex 2 to point b would change the orientation of the polygon; the vertex sequence 1-2-3 would change from clockwise to counterclockwise. This is generally undesirable.

The file `treemorph.blend` (in the same directory as the source code for the next sample program, `vids3d`) contains the palm tree created by deforming the cube.

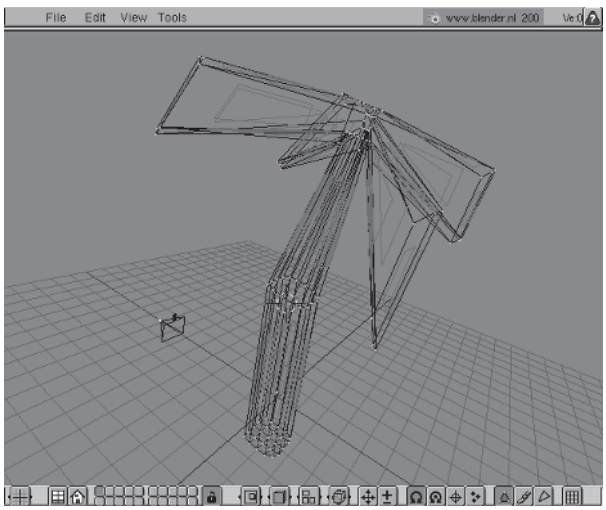


Figure 3-8: The completed palm tree as a compatible morph target to the original subdivided cube.

Next, let's look at applying a texture to the tree.

Applying a Texture and Assigning Texture Coordinates

We saw in Chapter 2 the theory of defining a texture space and applying this to a polygon. Chapter 6 covers using Blender in this way to define texture spaces and apply them to polygons. The problem is, this method of applying textures to polygons is not directly supported by Blender (actually, it is, but the results cannot be exported into a common file format), and requires us to use a separate program in addition to Blender. For now, we'll look at an alternative way of applying texture coordinates to polygons, which is supported directly by Blender and which can be exported and thus read into our programs. This alternative way of applying texture coordinates is a direct assignment of (u,v) texture coordinates to polygon vertices.



NOTE The following paragraphs describe the use of the (*u*,*v*) texture editor, a relatively new feature of Blender. This feature appears to have some small problems in Blender version 1.80, so you might want to use Blender 2.0 to do texturing.

Applying textures in this way requires us to use a special mode of Blender called FaceSelect mode. Toggle this mode on and off by pressing **f**. (You can think of this as being analogous to EditMode, which is toggled on and off with Tab.) In FaceSelect mode, right-clicking a face in the 3DWindow causes it to be selected. Press **Shift+right-click** to select multiple faces, **b** to select faces using border select mode, **a** to select/deselect all faces, or **h** to temporarily hide selected faces. Notice that these keys are the same ones that are used in EditMode for selecting vertices, or outside of EditMode for selecting objects. Blender draws the selected faces with a yellow dotted border.



NOTE In EditMode, you can select faces in a limited fashion by selecting all vertices belonging to the face. The problem with EditMode selection is that it selects all faces sharing the selected vertices; in other words, face selection in EditMode is actually an alternative interpretation of vertex selection, which sometimes makes face selection awkward in EditMode. On the other hand, FaceSelect mode allows you to selectively select or deselect faces at the face level, independent of the vertices.

You may find it easiest to use FaceSelect mode in the perspective view, rotating the viewpoint as needed to display the faces to be selected. Also, first hiding obstructing faces by pressing **Alt+h** is a useful way of making selecting easier.

After selecting faces in FaceSelect mode, Blender then allows you to map a texture to the selected faces in various ways. To do this, you first must load an image texture image. Do this in a window of type ImageWindow. It is easiest if you have an ImageWindow open next to the 3DWindow. Load an image via the Load button in the window header of the ImageWindow. Note that if the size of the ImageWindow is too small, the window header might be cut off, and the Load button might be invisible. Temporarily maximize the ImageWindow in this case to see the Load button. For the purposes of interactive texture mapping within Blender, the image should be square; its width and height must be identical. Supported file formats are JPEG, Targa, Iris, or HamX (a custom Blender format).



NOTE Note that Blender does not support the PPM format, which is the format used by l3d, so we need to have two copies of our texture images, one for Blender and one for l3d. You can use a program such as GIMP to convert between image formats. GIMP automatically recognizes file extensions, so converting from JPEG to PPM is really as simple as loading the JPEG and saving as PPM. Alternatively, the programs `cjpeg` and `djpeg` allow for batch mode conversion to and from the JPEG format.

Loading an image in the ImageWindow associates the texture image with the currently selected faces. Since our Videoscape plug-in object only supports one texture image per object, as mentioned above, you should select all faces and then load an image to associate the same image file with all faces. Do this as follows:

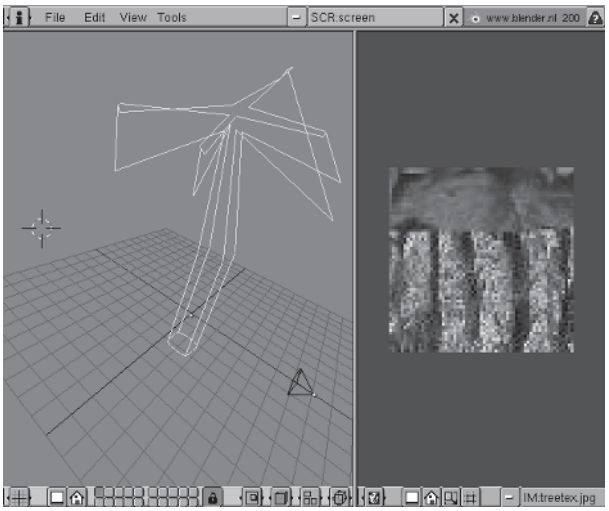


Figure 3-9: The ImageWindow.

1. Open an ImageWindow next to the 3DWindow.
2. In the 3DWindow, press **f** to enter FaceSelect mode.
3. If all faces are not selected, then press **a** until all faces are selected.
4. Press **Alt+z**. This turns on the textured display mode in the 3DWindow. Since we are going to be editing texture coordinates on the mesh, it is important that we see the results of our work immediately in the 3DWindow. (Type **z** to return to non-textured display and to toggle between normal wire-frame display and solid non-textured display.)
5. In the ImageWindow, click the **Load** button, and select an image file with square dimensions (width and height identical). Middle-click the selected filename to load it.

At this point, the texture image is loaded, and is mapped to every face on the polygon. The texture is mapped so that each face has a complete copy of the image. This is very similar to the default texture mapping used in the last sample program, along with the undesirable property that this is not in general the mapping that we want.

The way we change the mapping is as follows. In the 3DWindow, first select the faces for which you wish to apply the texture in a different way. Then, type **u**. This brings up a pop-up menu that allows you to specify the mapping of the current texture image onto the selected faces. The various entries in the menu have the following meanings.

- Cube: Applies a cubical mapping to the faces. From the object center, an imaginary cube (consisting of six faces) is drawn around the object. The texture mapped onto the faces takes on the orientation of one of the six faces of the cube which is oriented most closely with the face. Mostly vertical faces get a texture mapping defined by the vertical sides of the cube; mostly horizontal faces get a texture mapping defined by the horizontal (top or bottom) sides of the cube. Mathematically, either the (x,y), (x,z), or (y,z) coordinates are used as the (u,v) coordinates depending on the normal vector to the face. A cubical mapping is good for mapping textures onto mostly horizontal or vertical surfaces, such as walls, floors, and ceilings.

- Cylinder: Applies a cylindrical mapping. The texture space is an imaginary cylinder drawn around the object, and the texture coordinates are determined by projecting vertices of each polygon onto this cylinder. A cylindrical mapping is good for wrapping a label texture around a bottle.
- Sphere: Applies a spherical mapping. The texture space is an imaginary sphere around the object, and the texture coordinates are determined by projecting vertices of each polygon onto the sphere. A spherical mapping is good for wrapping a texture around a sphere or sphere-like object—a creature's head, for instance.
- Bounds to 64, bounds to 128: Uses the current projection of the polygons in the 3DWindow for computing the texture coordinates, rounding these to fit within a bounding box of 64 64 or 128 128 pixels, respectively.
- Standard 64, standard 128, standard 256: Uses the default square mapping, where every face is mapped to an identical square subset of the texture with a size of 64 64, 128 128, or 256 256 pixels, respectively. This is the initial face mapping.
- From window: Uses the current projection of the polygons in the 3DWindow for computing texture coordinates.

After selecting one of these mappings, Blender computes the mapping and stores a (u,v) texture coordinate with every vertex of every selected face. The results of the mapping are immediately visible in the 3DWindow (assuming that textured display mode has been activated with Alt+z).

Furthermore, the (u,v) mapping for the selected faces is displayed as a set of connected vertices in the ImageWindow. Remember that the ImageWindow displays the texture, which represents the texture space; the (u,v) texture coordinates for each vertex are locations in texture space, which can thus be displayed as points on the image. Within the ImageWindow, you can then use the normal mesh editing keys to select, grab, scale, or rotate the (u,v) coordinates within the ImageWindow, which has the effect of repositioning the point within texture space and thus assigning new texture coordinates to the original polygon's vertex. This allows you, for instance,

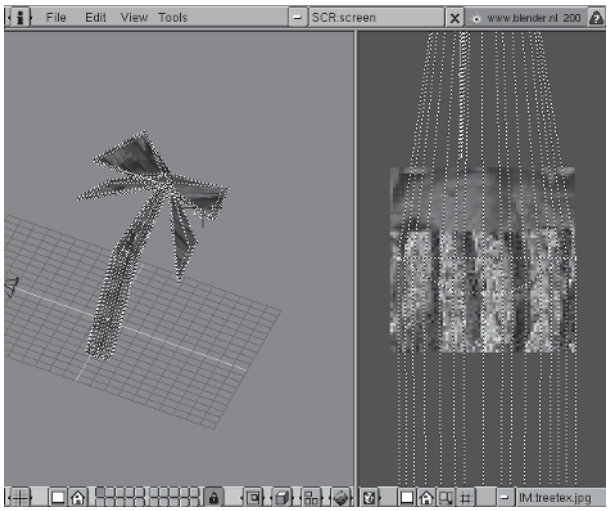


Figure 3-10: Texture coordinates in the ImageWindow (right), which correspond to the vertices of the selected faces (left) and can be moved arbitrarily.

to exactly map a certain part of the texture to fill an entire face, by moving the points corresponding to the face's vertices onto the appropriate locations on the image in the ImageWindow.

The texture file `treetex.jpg` (in the same directory as the `treemorph.blend` file) has been applied to the tree by using a cylindrical mapping for the trunk and a window-based mapping for the leaves. The texture file contains two parts. The top part of the texture is green, for the leaves; the bottom part of the texture is brown, for the trunk.

You can now see why we earlier stated that it is not necessarily very limiting that our Videoscape plug-in only allows one texture per object. By defining a suitably large texture containing many smaller images, and by assigning specific parts of the texture to specific faces, we can effectively apply several textures to different faces, even though physically we are only using one texture.

For this example of the palm tree, you can perform the mapping as follows:

1. Select all faces in the trunk of the tree model. You will need to rotate the model and extend the selection with **Shift+right-click** a few times to do this, because it is not possible to select a face when you are viewing it from its back side.
2. Press **u** and apply a cylindrical mapping to all the faces in the trunk of the tree.
3. In the ImageWindow, press **a** to select all texture vertices.
4. Press **s** to scale down the texture vertices vertically. Move the mouse cursor vertically towards the center of the selected vertices and middle-click. This limits the scaling in the vertical direction. Scale down the vertices until they are as high as the brown portion of the texture.
5. Press **s** to scale down the texture vertices horizontally. Move the mouse cursor horizontally towards the center of the selected vertices and middle-click. Scale down the vertices until they are as wide as the brown portion of the texture.
6. Press **g** to translate the texture vertices. Move them so that all vertices are located within the brown portion of the texture.

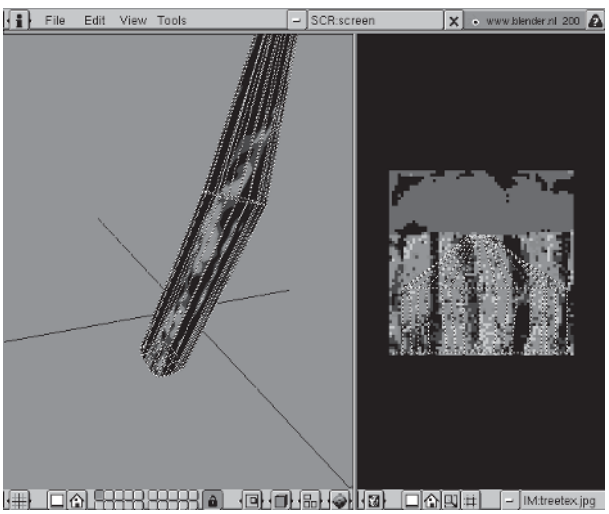


Figure 3-11: The faces for the trunk are cylindrically mapped to just the brown part of the texture image.

At this point, you have cylindrically wrapped the brown portion of the texture around the selected faces, which is a fairly good way of doing the mapping, since the trunk of a tree is essentially a cylinder.

Proceed in a similar fashion to wrap the green portion of the texture around the leaves, but use the From Window mapping. First, look at the tree from the side view in the 3DWindow so that the flat sides of the leaves are visible. Then, select all leaf faces, apply a texture mapping from the window view, and scale and translate the vertices in the ImageWindow to be positioned over the correct portion of the texture (the green part in this case).

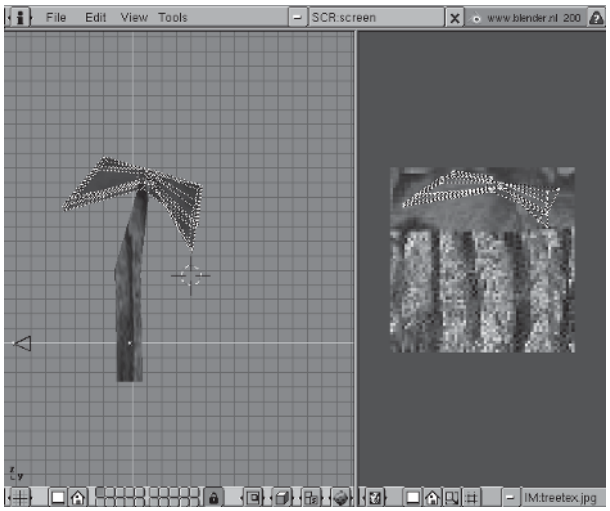


Figure 3-12

If you want to tweak the texture coordinates, you simply need to drag the appropriate vertices in the ImageWindow. Again, you would do this to exactly control which parts of the image get mapped to which parts of which faces. After repositioning a vertex in the ImageWindow, the 3DWindow automatically displays the results of the new mapping. To see the results of the mapping while you are still moving the vertex and before you have confirmed its new position, activate the Lock button in the header of the ImageWindow.



CAUTION Single-pass light mapping does not work if (*u*,*v*) coordinates have been directly assigned as described above.

It is important to note that manually assigning (*u*,*v*) vertices in this manner creates a non-linear mapping between texture space and world space. Each vertex in the ImageWindow represents a vertex of a polygon in 3D space. By manually positioning these vertices in the ImageWindow, we create a connection between the vertex location in 3D space and a (*u*,*v*) location in texture space. In other words, we create an *arbitrarily defined mapping* between world space and texture space. This means that with this method, we have no simple matrix that converts between texture and world space, as we did in Chapter 2, where we always used the texture space to calculate the texture coordinates as opposed to letting the user arbitrarily position them. The lack of a simple conversion between texture and world space when using direct (*u*,*v*) texture coordinate assign-

ment implies that any techniques requiring a texture-world space or world-texture space conversion cannot be applied. In particular, the single-pass light mapping strategy of Chapter 2, using a single pre-tiled surface combining texture and light map information, cannot be applied with objects that have been textured through direct (u,v) assignment, because this technique requires us to convert from texture (or lumel) space back into world space, which is not (easily) possible if the (u,v) coordinates have been arbitrarily assigned. Note that a two-pass light mapping strategy is still entirely possible even with arbitrarily assigned (u,v) coordinates, because in this case the second lighting pass has nothing to do with the arbitrary (u,v) coordinates, instead using a well-defined light map space. It's only with the single-pass light mapping that we have problems. For this reason, the Videoscape plug-in creates textured polygons, but not textured light mapped polygons.

Testing the Morph in Blender

At this point, you should have a textured tree object, which was formed by deforming a cube in the second morph target.

You can preview the morph in Blender simply by using the arrow keys to change the active frame number; as mentioned earlier, Blender automatically interpolates the vertices between the positions saved within the vertex keys. If you want to see the animation without keyboard intervention, press **Alt+a** within the 3DWindow. This animates the mesh automatically for a certain number of frames. Two buttons in the DisplayButtons control the starting and ending frames of the animation: the Sta button specifies the starting frame, and the End button specifies the ending frame. Press **Esc** to terminate the animation.

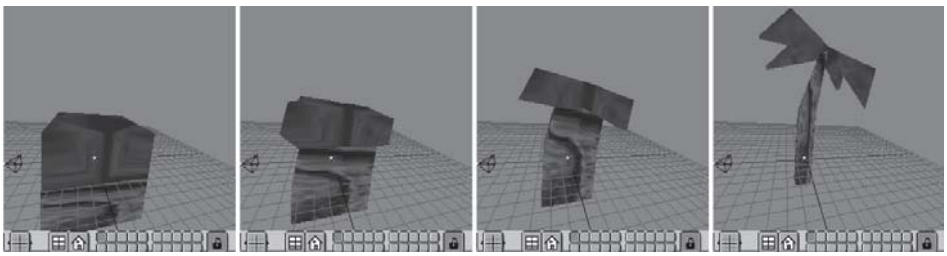


Figure 3-13: Testing the morph in Blender.

Exporting the Two Morph Targets

Once we are satisfied with the morphing animation, we can export the two (compatible) models as follows:

1. Change to frame 1. Notice that the mesh appears cubical in the 3DWindow, as defined by the first vertex key.
2. Press **Alt+w** and write the current mesh to the file `cube.obj`.
3. Change to frame 21. The mesh appears as defined by the second vertex key.
4. Press **Alt+w** and write the current mesh to file `tree.obj`.
 That was the easy part. Now comes the more tricky part—exporting the texture coordinates.

Exporting the Texture Information

The Videoscape file format does not store texture coordinates, so by exporting the morph target meshes to Videoscape, we lose the texture information. One simple solution is to export the file in VRML format, which is also an ASCII file format but is slightly more complicated to parse than Videoscape. With VRML export, Blender does save the texture coordinates in the VRML file. Fortunately, the ordering of the vertices and faces is the same in the Videoscape and the VRML files. This means that to associate texture coordinates with our Videoscape format objects, we can simply cut out the texture coordinate mapping within the VRML file, save it to a separate texture coordinates file, and read in this file later while reading in the Videoscape file.



NOTE The newest free version of Blender supports Python scripting (previously this was only available in the commercial C-key version). Therefore, it should also be possible to write a Python script to export the texture coordinates.

The texture coordinates file is the `mesh.uv` file we spoke of earlier when discussing the parameter string passed to the Videoscape plug-in object. It is simply an ASCII file, where each line contains a single (u,v) coordinate. The ordering of the lines in the texture coordinates file should be the same as the order of the vertex indices defining the faces in the Videoscape file. Note that the texture coordinates are associated with each vertex index defining a face, and not with the common vertices shared by all faces. This is because even though two adjacent faces may share a vertex, the texture coordinate at the shared vertex might be different for the two faces. Thus, texture coordinates are defined per face, and the texture coordinates file specifies one (u,v) coordinate for each vertex index of each face.

Again, you can create this texture coordinates file simply by cutting out the appropriate portion of a VRML exported file, as follows:

1. Save the file as VRML by pressing **Space** to bring up the Toolbox, choosing **File**, then **Save VRML**.
2. Edit the VRML file in your favorite text editor. Search for the beginning of the definition of your mesh object; the line begins with the keyword `DEF` (for definition) and contains the name of the mesh.
3. Search for the texture coordinate definition for this mesh. It follows the `DEF` line and begins with the keyword `TextureCoordinate2`. Following this line is a list of several texture coordinates.
4. Save the list of texture coordinates into a separate file. I suggest giving this file an extension of `.uv`, for instance, `meshname.uv`.
5. Specify this `meshname.uv` file in the parameter string passed to the Videoscape plug-in object, as discussed earlier. In this way, the (u,v) coordinates are read out of the file while loading the geometry from the Videoscape mesh file, and the texture mapping is applied.



CAUTION VRML export of texture coordinates on triangular faces does not appear to work in Blender. So, only use quadrilaterals if you wish to export texture information.

Team-Fly®

Unfortunately, it appears that there is a small bug in the Blender VRML export of texture coordinates with triangular faces. In the experiments I performed, the VRML exported texture coordinates are simply not correct if the faces are triangular. Making the faces into quadrilaterals (four-sided) seems to fix the problem. Thus, to export manually assigned texture coordinates in this manner, make sure all your faces have four sides.

Also, notice that the orientation of the *v* axis in texture space is different in Blender/VRML than it is in l3d. We have been using the convention that the upper-left pixel in the texture image is (0,0) in texture space. Blender's exported VRML texture coordinates, however, take the lower-left pixel as point (0,0) in texture space. This means that, for texture coordinates in the range 0.0 to 1.0 (which is the case if all of the texture coordinates lie within the image in Blender's ImageWindow), the *v* coordinate we use in l3d is 1.0–*v* from the `mesh.uv` file, which essentially flips the orientation of the *v* axis.

Importing the Morph Targets into a Program

The program `vids3d` imports our morph-compatible meshes and performs a morph between the cube and tree shapes.

Listing 3-1: File `shapes.h`, program `vids3d`

```
#include "../lib/geom/object/object3d.h"
#include "geom/vertex/verint.h"
#include "dynamics/plugins/plugenv.h"

class pyramid:public l3d_object {
    static const int num_keyframes = 2;
    int keyframe_no;
    l3d_two_part_list<l3d_coordinate> *keyframes[2];
    l3d_vertex_interpolator interp;
    bool currently_interpolating;

  public:
    pyramid(l3d_two_part_list<l3d_coordinate> *keyframe0,
            l3d_two_part_list<l3d_coordinate> *keyframe1);
    virtual ~pyramid(void);
    int update(void);
};
```

Listing 3-2: File `shapes.cc`, program `vids3d`

```
#include "shapes.h"

#include <stdlib.h>
#include <string.h>
#include "../lib/tool_os/memman.h"

pyramid::pyramid(l3d_two_part_list<l3d_coordinate> *keyframe0,
                 l3d_two_part_list<l3d_coordinate> *keyframe1)
    :
    l3d_object(100)
{
  keyframes[0] = keyframe0;
  keyframes[1] = keyframe1;

  currently_interpolating = false;
  keyframe_no=0;
```

```
}

pyramid::~pyramid(void) {
  //- set vertices pointer to NULL to prevent parent l3d_object from trying
  //- to delete these vertices, because they do not belong to us
  vertices = NULL;
}

int pyramid::update(void) {
  if(currently_interpolating) {
    vertices = interp.list;
    if(! interp.step()) {
      currently_interpolating = false;
    }
  }else {
    keyframe_no++;
    if(keyframe_no >= num_keyframes) {keyframe_no = 0; }
    int next_keyframe = keyframe_no + 1;
    if(next_keyframe >= num_keyframes) {next_keyframe = 0; }

    vertices = keyframes[keyframe_no];
    interp.start( *keyframes[keyframe_no], *keyframes[next_keyframe],
                  rand()%100 + 50, 3);

    currently_interpolating = true;
  }

  //- it is important to recalculate the normals because the mesa
  //- texture mapping reverse projection relies on correct normals!
  int i;
  for(i=0; i<polygons.num_items; i++) {
    polygons[i]->compute_center();
    polygons[i]->compute_sfcnormal();
  }
}
```

Listing 3-3: File `main.cc` program `vids3d`

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../lib/geom/object/object3d.h"
#include "../lib/geom/polygon/p3_flat.h"
#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rast3.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/geom/world/world.h"
#include "../lib/system/fact0_2.h"
#include "../lib/pipeline/pi_wor.h"
#include "../lib/dynamics/plugins/plugenv.h"
#include "../lib/geom/texture/tl_ppm.h"
#include "../lib/dynamics/plugins/vidmesh/vidmesh.h"
#include "../lib/tool_os/memman.h"

#include "shapes.h"

#include <stdlib.h>
```

```
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdarg.h>

class my_world:public l3d_world {
  public:
    my_world(void);
};

my_world::my_world(void)
    : l3d_world(400,300)
{
  l3d_screen_info *si = screen->sinfo;

  camera->VRP.set(0,0,-50,0);
  camera->near_z = float_to_l3d_real(5.5);
  camera->far_z = int_to_l3d_real(500);

  rasterizer_3d_imp->fovx = &(camera->fovx);
  rasterizer_3d_imp->fovy = &(camera->fovy);
  rasterizer_3d_imp->screen_xsize = &(screen->xsize);
  rasterizer_3d_imp->screen_ysize = &(screen->ysize);

  int i,j,k,onum=0;

  i=10; j=0; k=20;
  k=0;

  //- create two videoscape plug-in objects, one for the first morph
  //- target and one for the second

  l3d_object *morph_target0, *morph_target1;

  l3d_texture_loader *texloader;
  l3d_surface_cache *scache;
  texloader = new l3d_texture_loader_ppm(si);
  scache = new l3d_surface_cache(si);
  l3d_plugin_environment *plugin_env0, *plugin_env1;

  plugin_env0 = new l3d_plugin_environment(texloader, screen->sinfo, scache,
                (void *)"0 0 0 1 0 0 0 1 0 0 0 1 cube.obj tree.ppm tree.uv");
  morph_target0 = new l3d_object(10);
  //- max 10 fixed vertices, can be overridden by plug-in if desired
  //- by redefining the vertex list

  morph_target0->plugin_loader =
    factory_manager_v_0_2.plugin_loader_factory->create();
  morph_target0->plugin_loader->load("../lib/dynamics/plugins/vidmesh/vidmesh.so");
  morph_target0->plugin_constructor =
    (void (*)(l3d_object *, void *))
    morph_target0->plugin_loader->find_symbol("constructor");
  morph_target0->plugin_update =
    (void (*)(l3d_object *))
    morph_target0->plugin_loader->find_symbol("update");
  morph_target0->plugin_destructor =
    (void (*)(l3d_object *))
    morph_target0->plugin_loader->find_symbol("destructor");
  morph_target0->plugin_copy_data =
    (void (*)(const l3d_object *, l3d_object *))
```

```
  morph_target0->plugin_loader->find_symbol("copy_data");

if(morph_target0->plugin_constructor) {
  (*morph_target0->plugin_constructor) (morph_target0,plugin_env0);
}

plugin_env1 = new l3d_plugin_environment(texloader, screen->sinfo, scache,
              (void *)"0 0 0 1 0 0 0 1 0 0 0 1 tree.obj tree.ppm tree.uv");
morph_target1 = new l3d_object(10);
//- max 10 fixed vertices, can be overridden by plug-in if desired
//- by redefining the vertex list

morph_target1->plugin_loader =
  factory_manager_v_0_2.plugin_loader_factory->create();
morph_target1->plugin_loader->load("../lib/dynamics/plugins/vidmesh/vidmesh.so");
morph_target1->plugin_constructor =
  (void (*)(l3d_object *, void *))
  morph_target1->plugin_loader->find_symbol("constructor");
morph_target1->plugin_update =
  (void (*)(l3d_object *))
  morph_target1->plugin_loader->find_symbol("update");
morph_target1->plugin_destructor =
  (void (*)(l3d_object *))
  morph_target1->plugin_loader->find_symbol("destructor");
morph_target1->plugin_copy_data =
  (void (*)(const l3d_object *, l3d_object *))
  morph_target1->plugin_loader->find_symbol("copy_data");

if(morph_target1->plugin_constructor) {
  (*morph_target1->plugin_constructor) (morph_target1,plugin_env1);
}

//- create some pyramid objects
for(i=1; i<100; i+=20) {
  objects[onum=objects.next_index()] =
    new pyramid(morph_target0->vertices, morph_target1->vertices);

  *objects[onum] = *morph_target0;
  objects[onum]->num_xforms = 2;
  objects[onum]->modeling_xforms[0] = l3d_mat_rotx(0);
  objects[onum]->modeling_xforms[1].set
  ( float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.),
    float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.),
    float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.),
    float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.) );

  objects[onum]->modeling_xform=
    objects[onum]->modeling_xforms[1] |   objects[onum]->modeling_xforms[0];
  objects[onum]->vertices = morph_target0->vertices;

  l3d_screen_info_indexed *si_idx;
  l3d_screen_info_rgb *si_rgb;

  l3d_polygon_3d_flatshaded *p;
  for(int pnum=0; pnum<objects[onum]->polygons.num_items; pnum++) {
    p = dynamic_cast<l3d_polygon_3d_flatshaded *>(objects[onum]->polygons[pnum]);
    if(p) {
      p->final_color = si->ext_to_native
                       (rand()%si->ext_max_red,
                        rand()%si->ext_max_green,
```

```
                        rand()%si->ext_max_blue);
      }
    }

    if (objects[onum]==NULL) exit;
    objects[onum]->modeling_xforms[1].set
    ( int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(i),
      int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(1),
      int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(1),
      int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1) );
    objects[onum]->modeling_xform =
      objects[onum]->modeling_xforms[1] |
      objects[onum]->modeling_xforms[0] ;
  }

  screen->refresh_palette();
}

main() {
  l3d_dispatcher *d;
  l3d_pipeline_world *p;
  my_world *w;

  factory_manager_v_0_2.choose_factories();
  d = factory_manager_v_0_2.dispatcher_factory->create();

  w = new my_world();
  p  = new l3d_pipeline_world(w);
  d->pipeline = p;
  d->event_source = w->screen;

  d->start();

  delete d;
  delete p;
  delete w;
}
```

The program is almost identical to the morphing pyramid program from Chapter 2. First, we create two Videoscape plug-in objects and load the cube and the tree object into each of these objects. Then, we create a series of morphable pyramid objects, just as we did in Chapter 2. (The name pyramid is here somewhat of a misnomer, since the morphable objects are not pyramids in this case, but the naming remains the same to emphasize the similarity of the code with that of Chapter 2.)

In this case, instead of explicitly assigning the vertices in the source and destination morph vertex lists, we simply use the vertex lists from the already loaded plug-in objects. Then, each pyramid object morphs itself from its source to its target vertex list, and back again, using the vertex interpolator class.

The logic, therefore, is essentially the same as the simpler morphing program of Chapter 2, only now we are morphing between two real 3D models, created and textured within Blender. The logic is simple, yet the results look quite good in 3D.

Note that in this morphing program, in contrast to the morphing program of Chapter 2, we do recalculate the surface normals during morphing. This is because the Mesa texture mapping routines require a reverse projection, as we saw in Chapter 2. Performing this reverse projection

during Mesa texture mapping requires us to have an accurate surface normal vector, which is why we recalculate it whenever we interpolate the vertex positions.

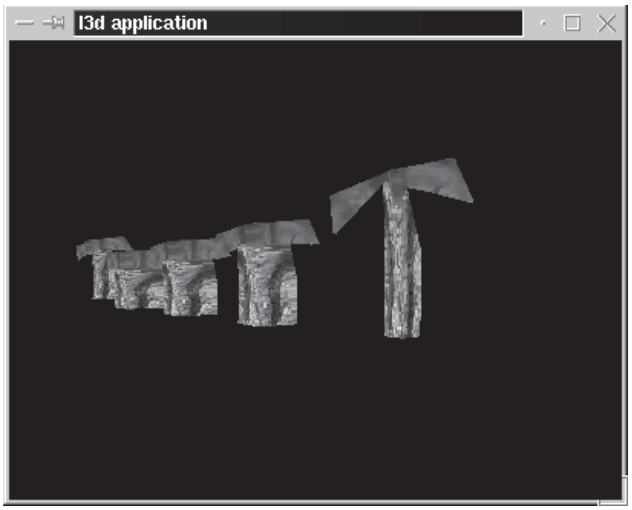


Figure 3-14: Output from sample program `vids3d`.

Tutorial: Using Inverse Kinematics and Rotoscoping to Model a Walking Human Figure

This tutorial illustrates two very powerful features of Blender: inverse kinematics and rotoscoping. These techniques allow us to animate a human-like jointed figure, which we then import into a program.

This tutorial is a small introduction to the field of *character animation*. Character animation is a special branch of animation dealing with animating *characters*—living entities such as humans, dogs, or imaginary creatures such as aliens. Even ordinarily inanimate objects, such as a sack of flour, can be animated as characters so that they appear to walk, move around, and come to life. Indeed, to animate literally means to bring to life. Character animation is a very broad field and requires years of study to master. This is because the subtle nuances of a character's movement convey much about that character's personality. Creating believable characters requires a firm concept of the character you wish to portray as well as an understanding of anatomy, proportion, staging, posing, timing, weight, movement, and so forth.

While this small tutorial cannot show you all of the aspects of character animation, what it does illustrate is the technical side of basic character animation using Blender. We'll create a simple human-like character and move its limbs around to simulate a walking motion. One of the most important tools for doing character animation is inverse kinematics, which we now look at in more detail.

Inverse Kinematics: Definition

Inverse kinematics, abbreviated *IK*, is best thought of as a way of automatically positioning a series of joints by specifying the position of just one point—the endpoint. An example will make this clear. Let's say we have modeled an arm object as a series of two segments: the top segment and the bottom segment.

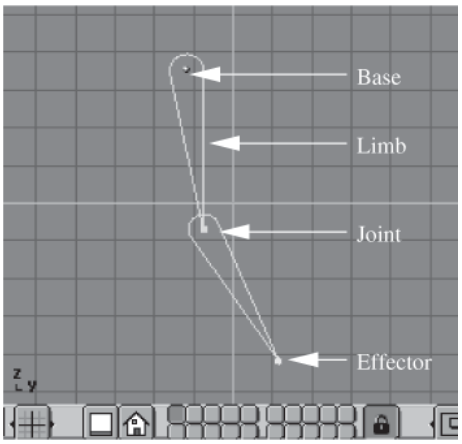


Figure 3-15: An arm modeled as a chain (or Ika) consisting of two bones (or limbs).

In Blender terminology, each of these segments is called a *limb*; in other literature, these segments are sometimes called *bones*. A series of connected limbs is called a *chain* or an *IK chain*. Blender also refers to a chain as an *Ika*, short for inverse kinematics. The connecting points between the chains are called *joints*. The start of the chain is the *root* or *base* of the chain. The last point in the chain is called the *effector.*

The important thing about an Ika chain is that moving the effector—a single point—leaves the base fixed in place, and automatically moves all points in between the base and the effector. In the arm example above, think of moving the effector as being equivalent to moving the hand. By moving the hand, which is located at the tip of the arm, we automatically cause the rest of the arm to be correctly repositioned. This makes animation in many cases much easier; we just pull the hand (effector) to the new position and the entire arm follows.

You can also move the base of an Ika chain. Moving the base of an Ika moves the entire chain and the effector, without changing the relative positions of the joints. You move the base of an Ika in order to reposition the entire Ika.

The alternative to inverse kinematics is *forward kinematics*, abbreviated *FK*. To position the arm with forward kinematics, you move the top part of the arm, which also moves the bottom part of the arm with it. Then, you move the bottom part of the arm, which then moves without affecting the top. This finally determines the position of the hand, located at the end of the arm. FK implies a simple hierarchical relationship between the top limb (the parent) and the bottom limb (the child). You can create such a simple hierarchical relationship for use in forward kinematics by pressing **Ctrl+p** (although not all hierarchical relationships are created for FK animation purposes).

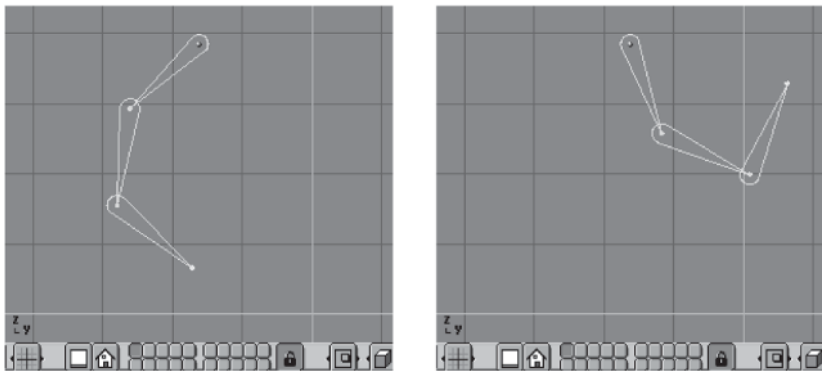


Figure 3-16: Moving the effector of the Ika chain causes the rest of the chain to automatically follow.

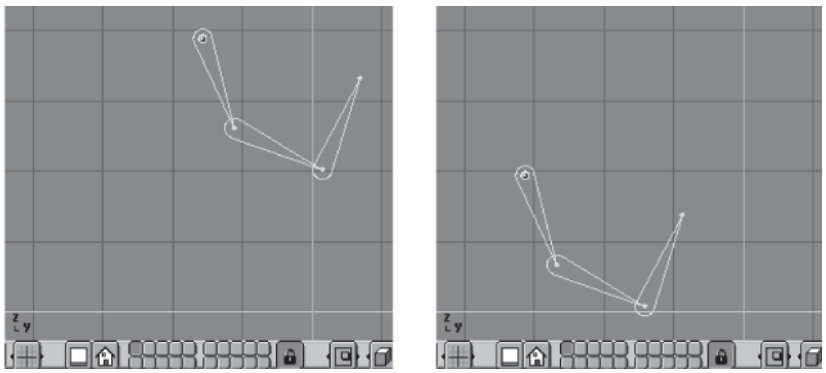


Figure 3-17: Moving the base of an Ika moves the entire Ika.

Just remember the following in order to distinguish between inverse and forward kinematics: In inverse kinematics, you move the lowest item in a hierarchy (the effector), which automatically successively moves all higher items in the hierarchy. In forward kinematics, you explicitly start at the highest level in the hierarchy, and successively manually move the lower items in the hierarchy, which finally determines the position of the lowest item in the hierarchy. Inverse kinematics works from the bottom up; forward kinematics works from the top down.

The choice of which is more appropriate—inverse kinematics or forward kinematics—depends on the most convenient way you like to think about the animation process. Generally, it's easier to think of limb animation in terms of inverse kinematics. For instance, when animating an arm, you usually think of the animation in terms of moving the hand (the effector) towards an object, and not in terms of moving the upper and lower arms individually. But since inverse kinematics automatically recalculates the joint positions, it also gives you less control over the exact animation. When animating a tail or a snake with many segments, it might be better to use forward kinematics.

For this tutorial, we'll use IK for animating the arms and legs of a human-like figure to create a simple walking motion. The first step is to understand how to create and manipulate Ikas in Blender.

Creating an Ika Chain in Blender

Let's begin modeling our human-like figure by creating four Ika chains: two for the arms and two for the legs. (The file `ika.blend`, located in the source code directory for the next sample program `ika`, contains the finished Ika model we create in the next few sections.) Begin with an empty Blender configuration (**Ctrl+x**). Then create the Ika chain for one leg as follows:

1. Position the 3D cursor as follows. Switch to front view (press **1** on the numeric keypad), and move the 3D cursor one unit to the right of center. Switch to side view (press **3** on the numeric keypad), and move the 3D cursor about two units above center.
2. Press **Shift+a** to add an item from the Toolbox, then select **Ika**. Blender adds an Ika with its base at the 3D cursor position.
3. Move the mouse cursor to position the effector of the Ika and left-click. The position of the Ika effector should be somewhat underneath and to the right of the Ika base. After left-clicking, notice that Blender then draws the next limb in the Ika automatically. The previous effector is now a joint between the first limb and the new limb; the tip of the new limb is now the new effector.
4. Move the mouse cursor to position the new effector. The effector should be somewhat below and to the left of the previous joint.
5. Middle-click to end the creation of the Ika chain.

It is important to make sure that the first limb points slightly forward, and the second limb points slightly backward; this will be the case if you position the effectors as described above. Positioning the limbs in this way creates a slight angle between the limbs. Later, when moving the effector, the angle between the limbs can never increase above 180 degrees. In other words, the Ika chain will never bend in the other direction of that specified when originally creating the chain.

Working with Ika Chains

As mentioned earlier, there are two ways to move an Ika chain. You can move the effector, which leaves the base in place and recomputes the position of all the joints between the base and effector. Or you can move the base, which moves the entire Ika to a new location.

Blender thus offers two movement modes for Ikas. A yellow dot indicates which movement mode is active. If the yellow dot is at the base of the Ika, then initiating a grab operation (press **g**) moves the entire Ika. If the yellow dot is at the tip of the Ika, then a grab operation moves the effector.

By default, the effector is active after creating an Ika. So, after creating the Ika as described above, press **g** and move the mouse cursor to see the effect of moving the effector. Press **Esc** to cancel the operation and return the effector to its original position. You could also left-click to accept the new position, but for now, the Ika should be positioned as we created it in the last section. This is because we want the Ika to be in a fairly straight position to start with.

Press **Tab** to switch the movement mode so that the base may be moved. This toggles between the two modes. With the base active, try grabbing the Ika and moving it around. Press **Esc** to cancel the operation when you are done (again, left-clicking would accept the new position).

Now, create the second Ika for the second leg as follows.

1. Ensure that the base of the Ika is active.
2. Switch to top view (press **7** on the numeric keypad).
3. Press **Shift+d** to duplicate the Ika. After copying the Ika, Blender immediately enters grab mode, as we saw earlier. This is the reason that it is important that the original Ika's base was active; if the effector was active, then the copied Ika's effector would also be active, meaning that the automatically initiated grab operation would reposition the effector and not the base, which is generally not what is wanted after duplicating an Ika.
4. Move the duplicated Ika one unit to the left of center and left-click. You should now have one Ika located one unit left of center and one located one unit right of center.
5. Make the effector of each Ika active since further manipulation of the Ikas takes place over the effector and not the base. Select each Ika and **Tab** until the yellow dot appears at the effector.

Creating the Arm Ikas

Creating the arm Ikas is similar to creating the leg Ikas. The main difference is that the arm Ikas should bend forward, whereas the leg Ikas bent backward. This means that when creating the arm Ikas, the first limb should point slightly backward, and the second should point slightly forward.

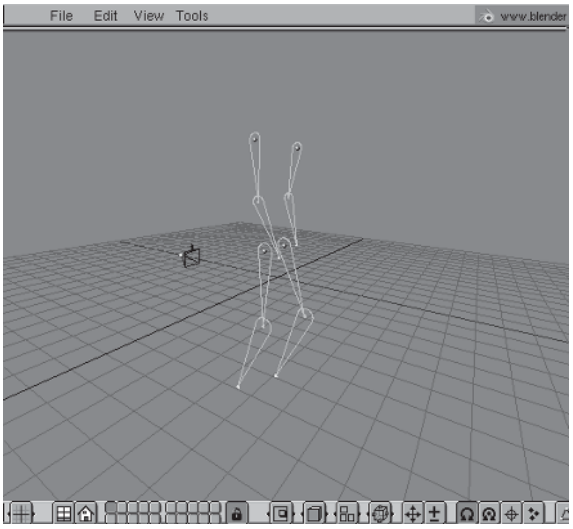


Figure 3-18

Create two arm Ikas. From the top view, the left and right arms should be about two units left and right of center, respectively. From the side view, the arm Ikas should be above the legs and stretch down slightly below the top of the legs. After creating the arm Ikas, activate the effector of each Ika by pressing **Tab**, because we later want to manipulate the effectors of the leg Ikas.

Team-Fly®

Creating the Main Body Ika

We have one last Ika to create. This is the Ika for the main body, and it consists of just one limb. Position the 3D cursor slightly between and slightly above the legs. Then, add an Ika which points upward, reaching up to approximately the height of the bases of the arm Ikas.

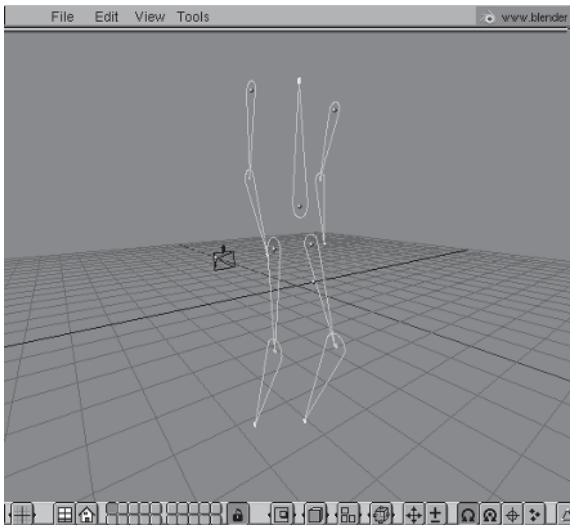


Figure 3-19

Parenting the Ikas into a Hierarchy

Next, let's parent the Ikas to one another in a hierarchical relationship. We make the main body Ika the parent of the arms and the parent of the legs. The idea is that moving the main body Ika should also move the child Ikas. The effector of the main body Ika can be thought of as the head of our character. In particular, when we move this effector, which is at the same height as the shoulders of our arm Ikas, this means that the bases of the arm Ikas also move. In other words, repositioning the head also automatically repositions the shoulders. We don't have to manually reposition the shoulders to match the new position of the head; an exact match would be almost impossible to achieve by hand anyway. This illustrates the power of Ikas in a hierarchy.

First, parent the arm Ikas to the body Ika as follows:

1. Select both of the arm Ikas by pressing **Shift+right-click**.
2. Extend the selection, again with **Shift+right-click**, to include the body Ika. The body Ika should now be active, displayed in a lighter color than the other two Ikas.
3. Press **Ctrl+p** to make the active object, the body Ika, the parent of the other selected objects, the arm Ikas. A pop-up menu appears with the choices Use Vertex, Use Limb, and Use Skeleton. These choices allow you to specify exactly which part of the Ika should become the parent. Use Vertex allows you to choose any point along the chain (the base, any of the joints, or the effector) to be the parent, and is what we use for this example. Use Limb means to use an entire limb, which we use in the next part of this tutorial. Use Skeleton means that several combined Ikas, called a *skeleton*, should be the parent. Using skeletons is unfortunately

beyond the scope of this book (but if you want to experiment, press **Ctrl+k** to create a skeleton from all selected Ikas, and then assign the skeleton as the parent of a mesh).

4. Choose **Use Vertex** from the parent menu. Another pop-up menu appears asking which vertex should be used as the parent. Notice that Blender at this point temporarily displays numbers next to the vertices of the Ika chain in the 3DWindow to assist in selection. Type **1**, since vertex 1 is the effector, and click **OK**.

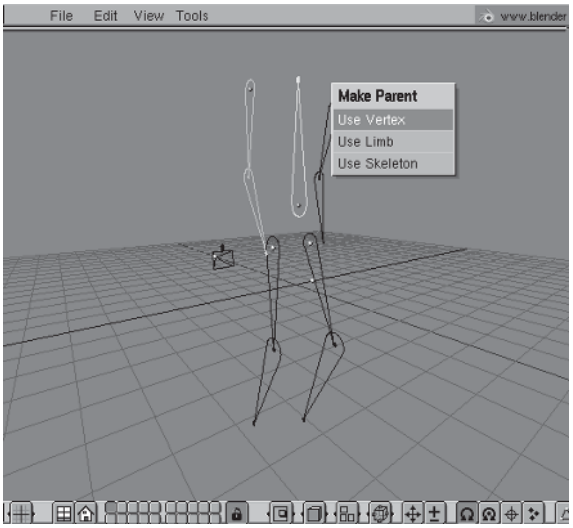


Figure 3-20

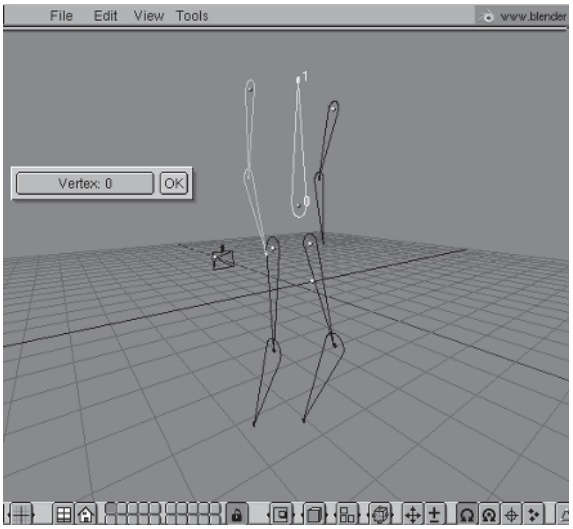


Figure 3-21

At this point, another pop-up menu appears asking about the children. The pop-up menu asks the yes/no question, Effector as child? If you click **OK**, the effector of the child Ikas will be hooked to the parent. If you press **Esc** to cancel the menu, the base of the child Ikas will be hooked to the parent.

5. Press **Esc** to link the bases of the child Ikas, as opposed to the effectors, to the parent.

Proceed similarly to assign the body Ika as the parent of the leg Ikas; only in this case, use vertex 0 (the base of the body Ika) as the parent. As with the arms, you should press **Esc** to the question Effector as child? because you want the bases of the leg Ikas to be hooked to the parent vertex 0 of the body Ika.

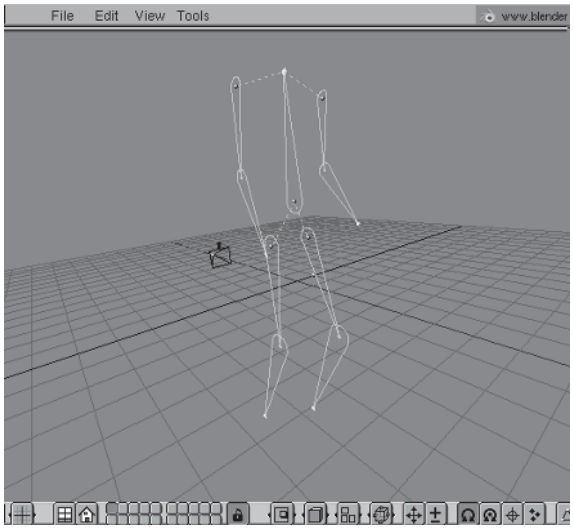


Figure 3-22

Testing the Ika Chains

With the Ika hierarchy built as described above, try moving around the effectors. Notice that moving the effector of the main body Ika allows you to cause your character to bend forward or backward; the shoulders follow the head. Notice also that moving the arm or leg effectors allows you to very easily place the hands and feet of your character.

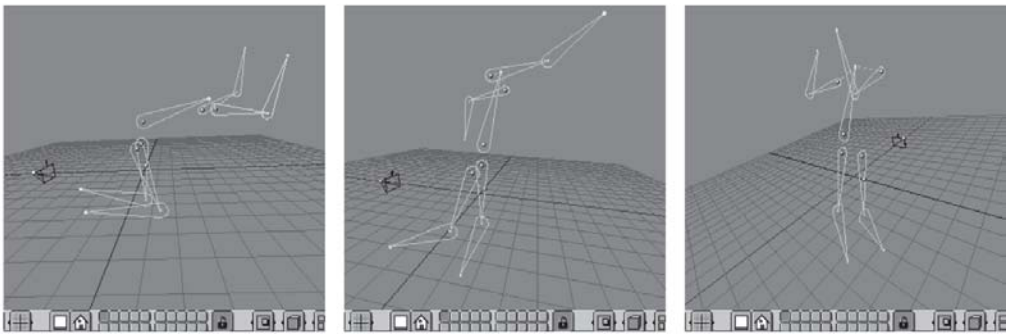


Figure 3-23: Some sample poses of the Ika hierarchy.

The next step is to animate the Ika chains to create a walking motion.

Animating the Ika Chains

Blender offers a number of ways of animating the Ika chains. We can specify key frames with certain positions of the effectors, we can draw the movement curves explicitly, or we can even record the motion of the mouse to allow us to move the limb in real time. Here, we use the first method, key frame animation.

Animating the Ika chains with key frame animation is almost identical to animating the vertices for 3D morphing as we did earlier. The procedure is as follows.

1. Choose a frame with the arrow keys.
2. Position all effectors for the frame. After positioning each effector, press **i** to insert a key frame, and select **Effector** from the pop-up menu. This saves the position of the effector.



NOTE Instead of manipulating and animating the effectors directly, you can also create an object (even an empty object, which is nothing more than a help object) which is the parent of the effector, and manipulate and animate the parent object instead of the effector. Transformations or animations on the parent object are then propagated to the child effector object. During key frame animation, you could then insert a LocRotSize key to save the location, rotation, and size of the parent object, instead of inserting an Effector key.

3. Change the frame number and repeat for all frames of the animation.

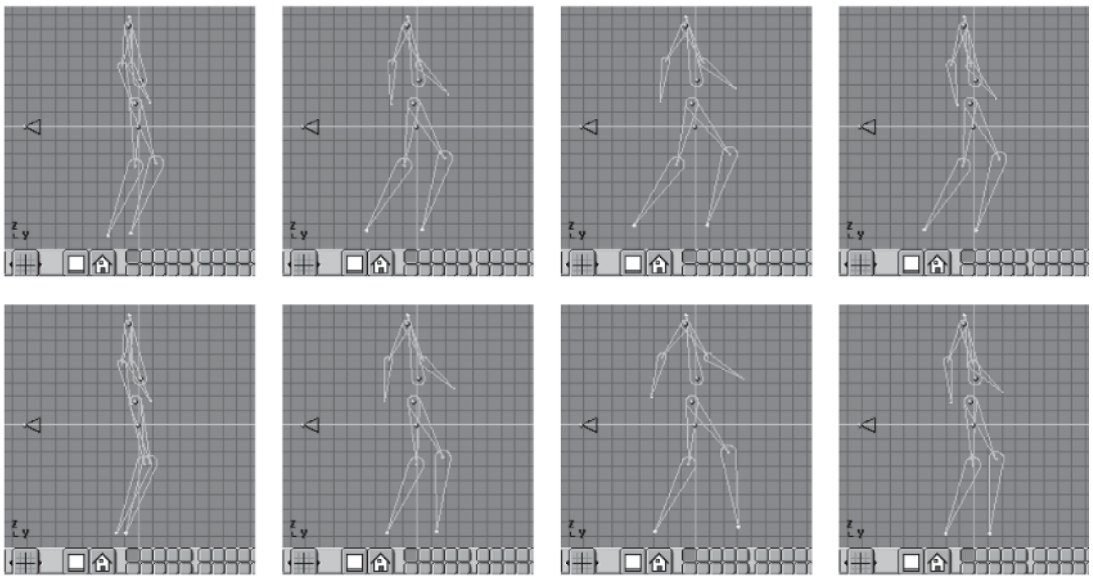


Figure 3-24: The key frames of the animation.

This is one of the most difficult parts of character animation—actually posing the character. For a simple walking motion, you can use the guideline that opposite arms and legs should move more or less in unison, and position the effectors appropriately for the key frames. But there is much, much more to creating a realistic walking motion—to say nothing of creating an realistic

and expressive walking motion. As mentioned earlier, this tutorial is only meant to introduce you to some of the important aspects of character animation. For now, just make the arms and legs swing back and forth over time.

Remember, you don't have to specify too many key frames; Blender interpolates the positions between the key frames to create a smooth animation. In this example, I created five key frames, at frames 1, 21, 41, 61, and 81. For each key frame, I positioned the effectors of the arm and leg Ikas, and inserted a key for each effector. Key frame 81 is identical to key frame 1, so that the animation—in this case, a walking motion—appears to loop.

Although it's not important for exporting the results (which we do later in this tutorial), it is a nice feature to make the animation loop in Blender. This way, you can let your animation run several times to see how it looks in a loop. You can do this by selecting an Ika, changing to the IpoWindow, and clicking on the W-shaped button, which represents a repeating angular curve. This button sets the extend mode for an interpolation curve to be cyclic; the curve then repeats its shape infinitely. Do this for all Ika curves by selecting each Ika in turn and setting its curves to be cyclically extended. In this way, the entire animation is then cyclic or looped. View the animation, as mentioned in the previous morphing tutorial, with **Alt+a**.

Now, we have a walking motion through our Ikas, but still have no polygonal model. The goal is to animate a polygonal model in a walking motion. Thus, the next step is to associate a series of meshes with the Ikas.

Connecting Ika Chains and Meshes

Ideally, we would like to create one single polygonal model with arms and legs, and associate each part of the model with its appropriate Ika—arm polygons with the arm Ikas and leg polygons with the leg Ikas. Unfortunately, this is not currently possible with Blender. In Blender, an Ika always affects an entire mesh, not just part of a mesh. So, we must split up our human-like model; in particular, the rule is: Each limb in each Ika must be associated with a separate mesh.

This rule ensures that the shape of each limb never changes. If we just had one mesh for several limbs (doing this would incidentally require us to create a skeleton with **Ctrl+k** as mentioned earlier), then this one mesh would be stretched and distorted based on the influence of several Ika limbs simultaneously. While this is fine for animating amorphous shapes (such as a sack of flour) without well-defined appendages, it looks rather ugly for shapes such as humans with arms and legs which shouldn't drastically change shape when they bend.

This rule, that we need a separate Ika for each mesh, means that we need two meshes for each arm, two meshes for each leg, and one mesh for the main body. We also add in an additional mesh for the head, even though it is not connected to any Ika in this example. Our humanoid figure thus consists of ten separate meshes.

For each Ika limb in the arm and leg Ikas, proceed as follows:

1. Create a new tube mesh, and position and scale it in 3D space so that it exactly surrounds the appropriate limb.
2. Select the tube and the Ika, with the Ika active.

3. Press **Ctrl+p** to parent the Ika to the mesh. Select **Use Limb** from the pop-up menu to make the Ika limb the parent. In the next pop-up menu, specify the desired limb number, which is the limb the tube surrounds.

For the main body, add a cube mesh and deform it so that it resembles a human torso. Then assign the (single) limb of the main body Ika to be the cube's parent. For the head, add a sphere object and specify the cube (the torso) as its parent.

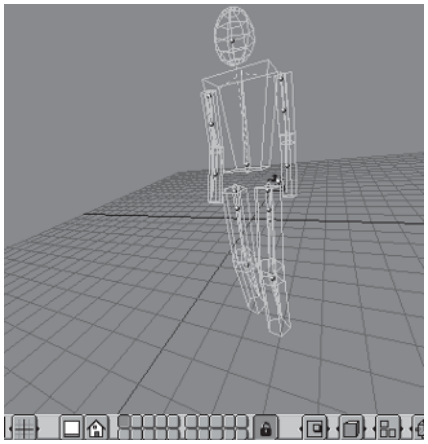


Figure 3-25

At this point, any movement of the limbs of the Ika (caused by movement of the effectors) should also propagate to the child tube objects. Try previewing your animation by pressing **Alt+a**; notice that the polygonal meshes now move along with their parent Ika limbs.

Texturing and Exporting the Model

Creating an IK animation in Blender that does not distort the mesh requires that we have a separate mesh for each Ika limb. But in a 3D program, it is more convenient to treat the entire character as one single mesh (unless the 3D application program itself directly supports IK, a topic we touch on briefly later in this chapter). We can easily join all selected meshes in the 3DWindow by pressing **Ctrl+j**, but the problem is that after joining meshes, the Ikas no longer correctly affect the joined mesh, meaning that all further frames of the animation are inaccessible. Undoing a join operation is also not possible. Therefore, copying a scene is the easiest way to export single, joined meshes for different frames of animation. In the copied scene, you join all meshes, and export the joined mesh. Then you delete the copy, and return to the original scene with the separate meshes. You change to the next frame of animation and repeat.

Thus, I recommend the following procedure to export the separate meshes as single, joined meshes:

1. Select all mesh objects.
2. Change to the first frame of animation you wish to export using the arrow keys.
3. Duplicate the entire scene. To do this, use the MENUBUT in the header of the InfoWindow. (In Blender 2.0, this button is located in the header of the ButtonsWindow when the

AnimButtons (activated with F10) are active. Click on the button, select **Add New** from the menu, and select **Full Copy** from the next menu. This creates an entire copy of the current scene, and switches to the copied scene.

4. Press **Ctrl+j** to join all selected meshes.
5. Press **Alt+w** to write the joined, single mesh into a Videoscape file. Choose a unique filename.
6. Delete the current scene by clicking on the **X** next to the **Scene** button. You return to the previous scene with the separate meshes.
7. Change to the next frame you wish to export, using the arrow keys. Repeat steps 3 through 6 for all frames you wish to export.

For this example, I chose to export eight key frames of animation, corresponding to frames 1, 11, 21, 31, 41, 51, 61, and 71. Notice that when positioning the effectors, we only specified four key frames, but we actually have 81 frames at our disposal, since Blender interpolates the effector positions between the frames we specified.

At this point, the exported meshes should all (hopefully) be morph-compatible, since they were formed by joining the same meshes in the same manner. Only the position of the individual sub-meshes has changed from frame to frame.

The last thing to do is to texture the model. You must texture the model after joining the meshes into a single mesh as described above. You only need to texture one of the models, not all of them, because the texture coordinate assignment across the model (which we save in the `mesh.uv` text file) stays the same regardless of how the model is posed; The texture file `ika.jpg` (located in the same directory as `ika.blend`) is a single 64 64 texture with five sub-images: a zigzag pattern, a smiley face, a white area, a green texture, and a skin texture. Remember that we need to convert the JPEG file to a PPM file when we import it into an l3d program.

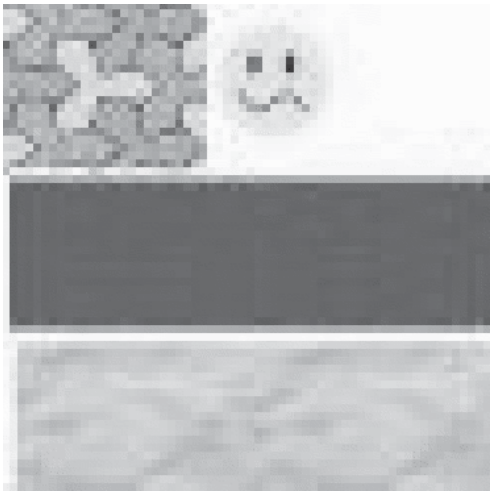


Figure 3-26: The texture file for the human-like model, containing five sub-images.

We map the various polygons of the model to the various sub-parts of the texture image as described in the 3D morphing example above. To recapitulate: open an ImageWindow and load an image; in the 3DWindow, enter FaceSelect Mode, select the desired faces, press **u**, select the desired mapping, and tweak the mapping by dragging vertices in the ImageWindow. For our human-like model, we use the following mappings:

- Front of head: Mapping From Window, when facing the front of the head in the 3DWindow. The resulting mapping is scaled in the ImageWindow to surround exactly the smiley face in the texture.
- Back of head: Mapping From Window, when facing the back of the head. The resulting mapping is scaled to surround the white part of the texture.
- Top of arm: Mapping Cylinder, scaled to fit within the zigzag pattern.
- Bottom of arm: Mapping Cylinder, scaled to fit within the skin pattern.
- Main Body: Mapping Standard 64, scaled to fit within the zigzag pattern.
- Leg: Mapping Cylinder, scaled to fit within the green pattern.

The textured, single-mesh model is located in the scene Textured within the file `ika.blend`. Make sure you activate textured viewing mode in the 3DWindow (**Alt+z**) to see the texturing in Blender. While it's not exactly a work of art, this model does illustrate the techniques used to generate textured, IK controlled meshes.

After applying the texture coordinates, export the file to VRML and save the texture coordinates section of the file to a file `ika.uv`, as described in the previous morphing example.

Importing the Textured Ika Meshes

Having exported the textured meshes from Blender, we can then import them into a program by using the Videoscape plug-in from the previous program. The next program, `ika`, does just this. The fact that the meshes were created with IK does not matter to the Videoscape plug-in; we just read them in as meshes and can then morph between them.

For this program, I just chose to morph back and forth between two key frames, which are saved in file `ika1.obj` and `ika7.obj`. These correspond to animation frames 11 and 71 in the original Blender file. Since we are just using two key frames in the program, this means that we are skipping over some of the eight key frames we exported. Since the motion in this example is so simple, this doesn't make much difference, but for more accurate motion, we could import more key frames into our program by loading the appropriate Videoscape files.



NOTE If you look closely, you will notice that the tips of the arms, which consist of triangular faces, have incorrectly assigned texture coordinates. This is due to the apparent VRML export bug mentioned earlier. To get around this, you should not use triangular faces with VRML export. I noticed this flaw in the model, but decided to leave it in so you could see what the problem looks like and how to recognize it.

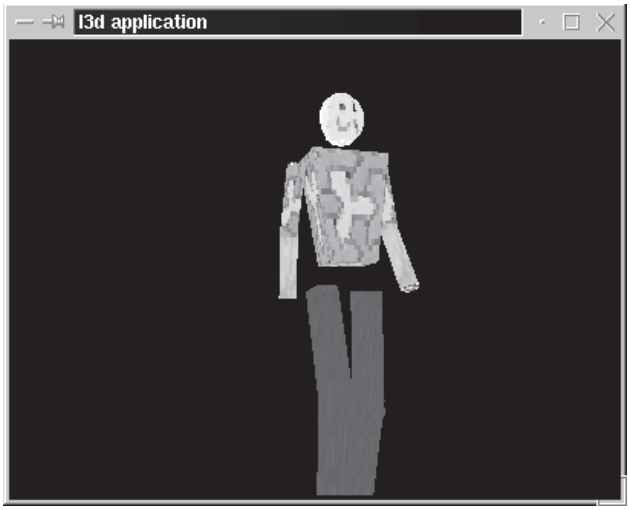


Figure 3-27: Output from program `ika`.

The program listing for this program is similar to the previous program. We load different Videoscape files, pass the texture coordinate file on to the Videoscape plug-in, make the morphing a bit faster, and only create one instead of several morphing objects, but the rest of the code is the same.

Listing 3-4: File `shapes.h`, program `ika`

```
#include "../lib/geom/object/object3d.h"
#include "geom/vertex/verint.h"
#include "dynamics/plugins/plugenv.h"

class pyramid:public l3d_object {
    static const int num_keyframes = 2;
    int keyframe_no;
    l3d_two_part_list<l3d_coordinate> *keyframes[2];
    l3d_vertex_interpolator interp;
    bool currently_interpolating;

  public:
    pyramid(l3d_two_part_list<l3d_coordinate> *keyframe0,
            l3d_two_part_list<l3d_coordinate> *keyframe1);
    virtual ~pyramid(void);
    int update(void);
};
```

Listing 3-5: File `shapes.cc`, program `ika`

```
#include "shapes.h"

#include <stdlib.h>
#include <string.h>

pyramid::pyramid(l3d_two_part_list<l3d_coordinate> *keyframe0,
                 l3d_two_part_list<l3d_coordinate> *keyframe1)
    :
    l3d_object(100)
{
  keyframes[0] = keyframe0;
  keyframes[1] = keyframe1;
```

```
   currently_interpolating = false;
  keyframe_no=0;
}

pyramid::~pyramid(void) {
  //- set vertices pointer to NULL to prevent parent l3d_object from trying
  //- to delete these vertices, because they do not belong to us
  vertices = NULL;
}

int pyramid::update(void) {
  if(currently_interpolating) {
    vertices = interp.list;
    if(! interp.step()) {
      currently_interpolating = false;
    }
  }else {
    keyframe_no++;
    if(keyframe_no >= num_keyframes) {keyframe_no = 0; }
    int next_keyframe = keyframe_no + 1;
    if(next_keyframe >= num_keyframes) {next_keyframe = 0; }

    vertices = keyframes[keyframe_no];
    interp.start( *keyframes[keyframe_no], *keyframes[next_keyframe],
                  30, 3);

    currently_interpolating = true;
  }

  //- it is important to recalculate the normals because the mesa
  //- texture mapping reverse projection relies on correct normals!
  int i;
  for(i=0; i<polygons.num_items; i++) {
    polygons[i]->compute_center();
    polygons[i]->compute_sfcnormal();
  }
}
```

TEAMFLY

Listing 3-6: File `main.cc`, program `ika`

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../lib/geom/object/object3d.h"
#include "../lib/geom/polygon/p3_flat.h"
#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rast3.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/geom/world/world.h"
#include "../lib/system/fact0_2.h"
#include "../lib/pipeline/pi_wor.h"
#include "../lib/dynamics/plugins/plugenv.h"
#include "../lib/geom/texture/tl_ppm.h"
#include "../lib/dynamics/plugins/vidmesh/vidmesh.h"

#include "shapes.h"
```

Team-Fly®

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdarg.h>

class my_world:public l3d_world {
  public:
    my_world(void);
};

my_world::my_world(void)
    : l3d_world(400,300)
{
  l3d_screen_info *si = screen->sinfo;

  camera->VRP.set(0,0,-50,0);
  camera->near_z = float_to_l3d_real(5.5);
  camera->far_z = int_to_l3d_real(500);

  rasterizer_3d_imp->fovx = &(camera->fovx);
  rasterizer_3d_imp->fovy = &(camera->fovy);
  rasterizer_3d_imp->screen_xsize = &(screen->xsize);
  rasterizer_3d_imp->screen_ysize = &(screen->ysize);

  int i,j,k,onum=0;

  i=10; j=0; k=20;
  k=0;

  //- create two videoscape plug-in objects, one for the first morph
  //- target and one for the second

  l3d_object *morph_target0, *morph_target1;

  l3d_texture_loader *texloader;
  l3d_surface_cache *scache;
  texloader = new l3d_texture_loader_ppm(si);
  scache = new l3d_surface_cache(si);
  l3d_plugin_environment *plugin_env0, *plugin_env1;

  plugin_env0 = new l3d_plugin_environment(texloader, screen->sinfo, scache,
                (void *)"0 0 0 1 0 0 0 1 0 0 0 1 ika1.obj ika.ppm ika.uv");
  morph_target0 = new l3d_object(10);
  //- max 10 fixed vertices, can be overridden by plug-in if desired
  //- by redefining the vertex list

  morph_target0->plugin_loader =
    factory_manager_v_0_2.plugin_loader_factory->create();
  morph_target0->plugin_loader->load("../lib/dynamics/plugins/vidmesh/vidmesh.so");
  morph_target0->plugin_constructor =
    (void (*)(l3d_object *, void *))
    morph_target0->plugin_loader->find_symbol("constructor");
  morph_target0->plugin_update =
    (void (*)(l3d_object *))
    morph_target0->plugin_loader->find_symbol("update");
  morph_target0->plugin_destructor =
    (void (*)(l3d_object *))
    morph_target0->plugin_loader->find_symbol("destructor");
  morph_target0->plugin_copy_data =
```

```
  (void (*)(const l3d_object *, l3d_object *))
  morph_target0->plugin_loader->find_symbol("copy_data");

if(morph_target0->plugin_constructor) {
  (*morph_target0->plugin_constructor) (morph_target0,plugin_env0);
}

plugin_env1 = new l3d_plugin_environment(texloader, screen->sinfo, scache,
              (void *)"0 0 0 1 0 0 0 1 0 0 0 1 ika7.obj ika.ppm ika.uv");
morph_target1 = new l3d_object(10);
//- max 10 fixed vertices, can be overridden by plug-in if desired
//- by redefining the vertex list

morph_target1->plugin_loader =
  factory_manager_v_0_2.plugin_loader_factory->create();
morph_target1->plugin_loader->load("../lib/dynamics/plugins/vidmesh/vidmesh.so");
morph_target1->plugin_constructor =
  (void (*)(l3d_object *, void *))
  morph_target1->plugin_loader->find_symbol("constructor");
morph_target1->plugin_update =
  (void (*)(l3d_object *))
  morph_target1->plugin_loader->find_symbol("update");
morph_target1->plugin_destructor =
  (void (*)(l3d_object *))
  morph_target1->plugin_loader->find_symbol("destructor");
morph_target1->plugin_copy_data =
  (void (*)(const l3d_object *, l3d_object *))
  morph_target1->plugin_loader->find_symbol("copy_data");

if(morph_target1->plugin_constructor) {
  (*morph_target1->plugin_constructor) (morph_target1,plugin_env1);
}

//- create some pyramid objects
objects[onum=objects.next_index()] =
  new pyramid(morph_target0->vertices, morph_target1->vertices);

*objects[onum] = *morph_target0;
objects[onum]->num_xforms = 2;
objects[onum]->modeling_xforms[0] = l3d_mat_rotx(0);
objects[onum]->modeling_xforms[1].set
( float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.),
  float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.),
  float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.),
  float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.) );

objects[onum]->modeling_xform=
  objects[onum]->modeling_xforms[1] |   objects[onum]->modeling_xforms[0];
objects[onum]->vertices = morph_target0->vertices;

l3d_screen_info_indexed *si_idx;
l3d_screen_info_rgb *si_rgb;

l3d_polygon_3d_flatshaded *p;
for(int pnum=0; pnum<objects[onum]->polygons.num_items; pnum++) {
  p = dynamic_cast<l3d_polygon_3d_flatshaded *>(objects[onum]->polygons[pnum]);
  if(p) {
    p->final_color = si->ext_to_native
                     (rand()%si->ext_max_red,
                      rand()%si->ext_max_green,
```

```
                        rand()%si->ext_max_blue);
    }
  }

  if (objects[onum]==NULL) exit;
  objects[onum]->modeling_xforms[1].set
  ( int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(i),
    int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(1),
    int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(1),
    int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1) );
  objects[onum]->modeling_xform =
    objects[onum]->modeling_xforms[1] |
    objects[onum]->modeling_xforms[0] ;

  screen->refresh_palette();
}

main() {
  l3d_dispatcher *d;
  l3d_pipeline_world *p;
  my_world *w;

  factory_manager_v_0_2.choose_factories();
  d = factory_manager_v_0_2.dispatcher_factory->create();

  w = new my_world();
  p  = new l3d_pipeline_world(w);
  d->pipeline = p;
  d->event_source = w->screen;

  d->start();

  delete d;
  delete p;
  delete w;
}
```

Rotoscoping and Inverse Kinematics

Blender offers an interesting feature which can make creating IK animations easier. This feature is called *rotoscoping*. Rotoscoping is the use of live video material to assist in creating computer animations. Blender's support for rotoscoping allows individual frames of a movie file to be synchronized with the Blender animation frame. The movie image is superimposed on top of the 3DWindow. Changing the Blender animation frame with the arrow keys also changes the displayed frame of the movie. For instance, moving ahead five animation frames in Blender also moves ahead five frames in the movie.

The idea is that the movie footage shows the movement of a character (e.g., a human) from a fixed camera position. Then, the 3D camera in Blender should be placed in a position in space corresponding to the position of the real movie camera used to take the video footage. The virtual camera and the real camera then view the scene from the same position. Rotoscoping work is thus done in Blender in the camera view (key **0** on the numeric keypad). For each frame of animation, we position the Ikas such that the apparent position of the Ikas in Blender's camera view is identical to the apparent position of the real actor's arms and legs in the real movie footage. This

alignment of the Ikas is made possible because the correct frame of the movie image is superimposed upon the 3DWindow. We align the Ikas with the real movie footage for one frame, change to the next frame, align the Ikas again, and so forth. In this manner, the apparent movement of the Ikas exactly matches the apparent movement of the real actor, frame for frame. After synchronizing the movement in this way, we are then free to view the animation from any angle or to export the animation as a series of meshes for vertex morphing in a 3D program.

To use rotoscoping in Blender we must load a movie as a background image for the 3DWindow. Do this as follows:

1. Add a help object, such as a plane. This help object will use a material, which in turn will use a texture, which in turn will hold the movie file.
2. Press **F5** to change to the MaterialButtons. Add a new *material* through the MENUBUT button in the header of the ButtonsWindow. A material specifies visible properties of a surface and is valuable for creating realistic renderings with Blender; however, we are just using the material as a placeholder for a texure.
3. Press **F6** to change to the TextureButtons. Add a new texture through the MENUBUT button in the header of the ButtonsWindow. The new texture is connected to the previous material.
4. Activate the TOGBUT Image near the top of the ButtonsWindow. This sets the texture type to be an image file. Blender supports two kinds of movies for rotoscoping: single file movies with several frames in one file (such as an AVI file), or movies saved as a series of sequentially named separate files (e.g., `file.001.jpg`, `file.002.jpg`, `file.003.jpg`, and so forth). If you want to use a single file movie, activate the TOGBUT Movie; otherwise, leave it unactivated.
5. In the NUMBUT Frames, enter the total number of frames in the animation. For multiple file movies, this specifies the number of files to be sequentially loaded. For single file movies, this number specifies how many frames in the animation will be used. If you do not set this number, you will see the first image of the movie, but no further ones—so be sure to set this value.
6. Click the button **Load Image**, and select either an AVI movie file or the first numbered file of a series of images. Blender loads the animation file(s) into the texture.
7. In the 3DWindow, press **Shift+F7** to access the ViewButtons for this 3DWindow. The window changes from type 3DWindow to type ButtonsWindow, and the ViewButtons are displayed.
8. Activate the TOGBUT BackgroundPic. A MENUBUT appears beneath the TOGBUT; use it to select the movie you loaded earlier.
9. The bottommost MENUBUT in the ViewButtons controls the use of a movie as a background picture (as opposed to a still image). To use a movie as a background picture, select the texture, which you previously created and into which the movie was loaded, with this MENUBUT. A TEXTBUT appears beneath the MENUBUT displaying your selection.



NOTE Once, I encountered what may be a bug in Blender 2.0, which prevented the use of the MENUBUT to select a movie texture. The symptom was that selection of the texture in the MENUBUT never caused the TEXTBUT to appear, indicating that Blender did not accept my selection. To solve this, I deleted the file `.B.blend` from my home directory, which saves Blender's default configuration information. After deletion of the defaults file, the function worked again; evidently, some odd configuration prevented use of the movie texture.

10. Press **Shift+F5** to change the window back to a 3DWindow.
11. Press **0** on the numeric keypad to switch to camera view. You should now see the movie superimposed on top of the 3DWindow. Additionally, in top, front, and side orthogonal views, you also see the movie. For any other viewpoint, you do not see the movie. This indicates that rotoscoping is only useful in orthogonal or camera views. For the purposes of character animation, camera view is the view to use. (For tracing 2D logos, orthogonal mode could be used.)

Now, whenever you change the Blender animation frame with the arrow keys, the superimposed movie also changes its frame. So, to do rotoscoping, you simply:

1. Build an Ika hierarchy and parent it to a series of meshes.
2. Position the 3D camera in the same position and orientation relative to the character as in the original film footage. Either you must know this information, or you can guess it by looking at the footage. For instance, it might appear that the camera is slightly above the character and is a few meters distant. The more accurately you place the camera, the better results you will get.
3. Use the arrow keys to change to the first frame of the animation and of the movie.
4. Switch to camera view.
5. Position the Ikas (through their effectors) so that the profile of the surrounding meshes matches the profile of the character on the movie frame. You may find it helpful to have several 3DWindows open simultaneously: one in camera view to monitor the camera view of the Ikas, and some orthogonal view 3DWindows for precise positioning of the Ika effectors in side, front, and top view.
6. Insert keys for all Ika effectors (or their parent objects, if you have chosen to manipulate effectors through parent objects). Additionally, if you have rotated the Ikas or their parent objects, you should also insert keys to save the rotation (type **Rot**, **LocRot**, or **LocRotSize** to save the rotation).
7. Change to the next frame of animation. Depending on how precisely you wish to mimic the animation, and how much time you are willing to invest, you may want to synchronize every frame, which is the most exact method, or you may work in steps of, say, ten frames, allowing Blender to interpolate the rest of the movement. Repeat the synchronization steps 5 and 6 for all frames of animation.

You can incrementally work on synchronizing the frames. For instance, you could start by synchronizing the positions of the Ikas once every 30 frames, to get a rough capture of the gross motion. Then, preview the animation and observe if the automatically interpolated Ika positions match up closely enough with the video footage. If not, then resynchronize the Ikas every 15

frames at those points where the interpolated position deviates too greatly from the actual position. Continue this process to achieve the desired accuracy.

Programming IK and FK

Until now, we've looked at IK from the point of view of a 3D modeling program; we manually pose our character in the 3D modeler with the help of IK, then export the results to meshes which we then animate in our program. A more comprehensive approach, and one which has started to become more common in recent years, is actually to do the IK calculations in real time in the 3D program itself. Essentially, you need to create a hierarchical data structure to store the limbs in an IK chain, and a system of solving for all intermediate positions and orientations of the intermediate joints given the position of the base and the effector. Indeed, Blender itself must perform exactly such a computation to position the Ikas. The techniques unfortunately go beyond the scope of this book, and are sometimes combined with physically based simulation so that movement of the character's joints obeys the physical (Newtonian) laws of motion. Creating animations in this manner can create very realistic, non-repetitive animations; with a key frame approach, all animations are pre-recorded and appear the same whenever they are played back within the program. However, physically based systems also give less control over exactly what sort of motion appears, since it is automatically calculated; this can be disadvantageous if you want to model specific nuances of motion.

Another reason for programming IK into the 3D application program itself is to allow more sophisticated ways of handling the appearance of polygons at joints. With IK under program control, we can dynamically create smoothing polygons at the joints, depending on the position of the limbs, so that elbows and knees don't appear to be two separate sticks glued together, but instead smoothly flow into one another, like real skin on top of bones. We can also cause other dynamic position-based effects, such as bulging muscles when an arm is flexed.

Essentially, computing the IK beforehand makes the 3D program simpler and allows for easier control of the exact appearance of the motion, but computing the IK in the program allows you more opportunities to interact with the IK computation and its effects on the mesh. The availability of good tools is the key to combining the advantages of precomputed and program-controlled IK.

Even without programming a complete IK system, you can still program a simple limb-based (or bone-based, depending on your terminology) system quite easily, for the sole purpose of saving memory [LUNS98]. Consider that for each frame of our IK-generated animations, we export the entire mesh data, with positions of all vertices. Thus, for each frame of animation, we store the positions of all vertices in a vertex list in memory. For large animations, this can be a lot of memory. We can save memory by recognizing that the model is not a loose collection of unrelated vertices, but is instead a grouping of limbs which just happen to be combined into one mesh for convenience. Thus, even if all vertices move from one frame to the next, certain relationships among the vertex positions remain constant—specifically, all vertices belonging to a limb remain in the same position relative to one another. It is only the position of the limbs relative to one another that changes from frame to frame.

Thus, a limb-based animation system stores each limb as a set of vertices in memory, and stores the movement data in terms of the movement of the limb (which is one transformation matrix applied to every vertex of the mesh associated with the limb) instead of in terms of explicit vertex positions for all vertices in the limb. Writing such a system requires you to extract the limb information from a raw mesh after it has been exported—a tricky process, but not impossible. First, you find all limbs by finding vertices whose relative positions do not change from frame to frame. This divides the mesh up into a set of limbs. Then, given the raw vertices for a limb at the previous frame and at the next frame, you must derive a transformation matrix converting from the old vertex positions to the new vertex positions. You can do this by forming a coordinate system based on certain vertices of the original and the new positions of the limb, then using a matrix to convert between the coordinate systems.

Note that these limbs and the transformation matrices moving these limbs can all be generated completely automatically. With some additional manual intervention, it is also possible to arrange the limbs—and their transformation matrices—into a hierarchy. For instance, movement of the forearm limb would not be viewed in terms of a forearm-to-world-space matrix, but instead in terms of the series of matrices forearm-to-upper-arm, upper-arm-to-shoulder, and shoulder-to-world-space. The process for automatically extracting limbs from an unstructured mesh has no way of deciding on a hierarchy of limbs, which is why this information would need to be specified additionally.

Once we have specified this additional hierarchy, the entire animation system essentially becomes a forward kinematics system. The original positions of the mesh vertices may have been generated via IK, but this information was not exported; only the raw mesh geometry was exported. Through the limb extraction process, we obtain collections of vertices which move together, then derive their orientations and arrange them manually into a hierarchy of limbs and their transformation matrices. The hierarchy then controls the mesh in an FK manner: the precomputed transformations from the parent limbs propagate to the child limbs. The result is an animation that results in the same mesh animation we would achieve with vertex morphing, but that uses less memory.

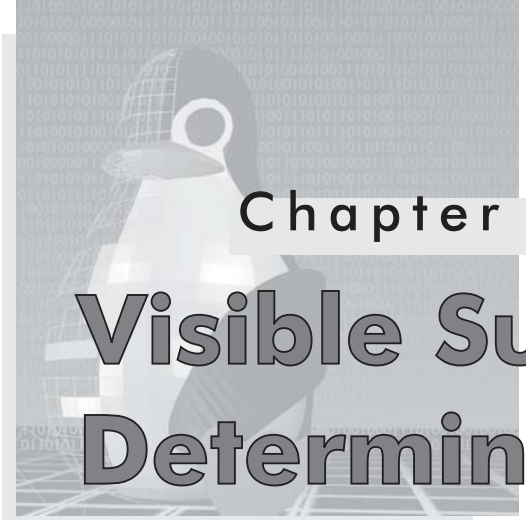
An example of how such a limb-based system can save memory is animating a walking motion [SEGH00]. To animate a walk using vertex morphing, we need at least three keys for the feet (back, centered, forward); any fewer and it becomes obvious that the feet are translating in a straight line from back to front, thus shortening the legs. To look good, we need at least five keys. With a limb-based hierarchial system, we just need two keys (back, forward) since the points will rotate around the joint properly, maintaining their shape with respect to the limb.

Summary

In this chapter, we looked at a number of useful Blender modeling techniques. We saw how to use the Videoscape file format in combination with additional files for storing a texture image and texture coordinate information, which we then read in using a Videoscape plug-in object. We saw how to use Blender's VRML export for extracting the texture coordinates. We also looked at the animation topics of compatible morph targets, inverse kinematics, and rotoscoping.

We now know quite a bit about Linux 3D graphics programming—both theoretical and practical aspects. With the free compilers and 3D modeling tools available for Linux, you can put all of this knowledge to work by practicing writing 3D programs and creating interesting, animated, textured 3D models for use in these programs.

Starting with the next chapter, we turn our attention to some of the 3D techniques specific to creating larger, more interesting, more interactive environments. The next chapter focuses on general visual surface determination techniques used to deal with larger 3D datasets.



Chapter 4

Visible Surface Determination I: General Techniques

We have seen many facets of Linux 3D graphics programming: operating system issues, the X Window System, 2D graphics, 3D geometry and mathematics, classes for organizing 3D world data, 3D modeling and texturing, import and export of polygonal models, and animation. Starting with this chapter, we begin to put all of this information together, to create larger, interactive 3D environments. In this and the coming chapters we'll look at techniques for handling larger datasets, tools for creating these datasets, and some miscellaneous topics which make for a more realistic and enjoyable 3D experience (collision detection, sound, and basic networking).

This chapter deals with the topic of *visual surface determination*, abbreviated VSD, which is the determination from an arbitrary set of polygons which ones are visible for the given viewpoint. VSD is a large area of study, so we devote two chapters to the topic. In this chapter, we cover generally applicable VSD techniques and principles, which can be used in most situations without much difficulty. In particular, we discuss the following topics:

- Purpose and goals of VSD algorithms
- Back-face culling
- Hierarchical view frustum culling and bounding volumes
- Painter's algorithm
- *z* buffer algorithm

Each of the discussed algorithms is also implemented in l3d library classes to illustrate exactly how the algorithm works in practice.

The Goals of VSD

VSD algorithms have two main goals: to ensure a correct display and to efficiently reduce the amount of computation which must be done. Let's discuss each separately.

First, the correctness issue addressed by VSD algorithms aims to ensure that farther polygons do not obscure nearer polygons, since this is a physical impossibility due to the properties of light. Drawing polygons to the screen in no particular order often leads to an incorrect display where farther polygons sometimes obscure nearer ones. For this reason, the l3d library, as we have seen it thus far, implements a simple version of the so-called painter's algorithm, which we return to in more detail later in this chapter. Other VSD algorithms we look at in this chapter take different approaches but have the same goal: each pixel drawn to the screen should, ideally, correspond to (that is, result from the projection of) a point that we are physically able to see from our 3D model for the given viewpoint. We say "ideally" because VSD algorithms for interactive 3D environments can often get away with slightly imperfect results; since the viewer is probably moving, any small imperfections in the VSD for the current frame may very well disappear for the next frame. The situation is different with photorealistic, non-real-time algorithms, where every rendered image should be as close to perfect as time, algorithms, and resources allow.

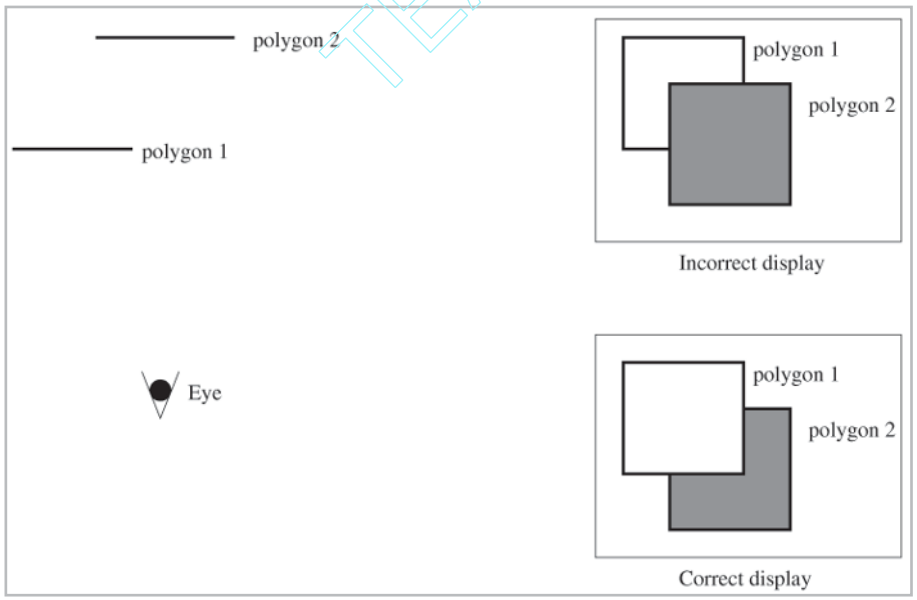


Figure 4-1: The correctness issue of VSD. Farther polygons or parts of polygons cannot obscure nearer polygons. A correct display mimics physical viewing reality.

Second, the efficiency issue addressed by VSD algorithms aims to reduce the amount of work that must be done when drawing a scene, trying to restrict processing of polygons to just those polygons that are indeed visible for the current viewpoint. Efficient visible surface determination is important, because for a large world consisting of many polygons, generally only a small

Team-Fly®

fraction of these are visible. Many are invisible because they are behind the viewer, pointing away from the viewer, too far from the viewer, outside the field of view, or obstructed by other polygons. Sequentially checking every single polygon in the environment to see if it is visible or not can be extremely time consuming if most of the polygons are indeed invisible; in the end, we only need to draw the visible polygons. Thus, visual surface determination techniques try to use efficient data structures to eliminate processing of as much of the world as possible, and to focus processing on only the parts of the world that can be seen. The main obstacle to developing efficient VSD algorithms for interactive environments is the dynamic nature of the environment: both the viewer and the environment may move or change from frame to frame, meaning that the VSD problem essentially needs to be resolved any time anything in the environment changes. We can, however, reuse visibility information from the previous frame in solving the VSD problem for the current frame, since much of the information as to which polygons were visible and invisible may still be valid if the viewer and/or environment only change slightly from frame to frame. Additionally, we can try to precompute some viewpoint-independent visibility information, which we statically store then reference at run time to solve the VSD problem for the current viewpoint. Examples of such precomputed visibility information include portals and BSP trees, which we cover in the next chapter.

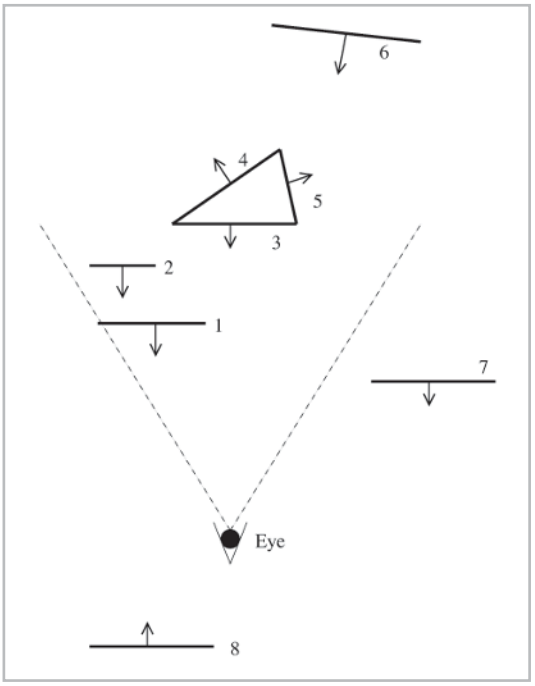


Figure 4-2: The efficiency issue addressed by VSD. (Arrows represent polygon surface normals.) Only polygon 1 and part of polygon 3 are visible; the rest are invisible because they are behind the viewer (8), pointing away from the viewer (4, 5), too far from the viewer (6), outside the field of view (7), or obstructed by other polygons (2, part of 3). Processing as few invisible polygons as possible is an efficiency goal of VSD. If the viewpoint or the polygon positions change, so does the visibility information.

The opposite of visual surface determination is *hidden surface removal*, abbreviated HSR. This refers to the removal of surfaces which are hidden or not visible. Thus, we use the terms VSD and HSR more or less interchangeably; they both refer to the same problem, just seen from different viewpoints.

Recall the complementary roles of hardware acceleration and VSD schemes. The general philosophy is "cull what you can, accelerate what you can't." In other words, in the presence of accelerated hardware, culling or VSD schemes can stop processing once the polygon counts per frame lie within the ranges that can be handled by the hardware. For instance, if the hardware can draw several hundred thousand polygons in 1/60th of a second, then we don't need to painstakingly compute exactly which edges or pixels of one polygon obscure which edges or pixels of another polygon; we simply send all of the "probably visible" polygons to the hardware, and let the hardware draw them all. Even if we waste time drawing invisible polygons, this is hardware-accelerated wasted time, which takes less than 1/60th of a second. As you read this chapter, keep in mind that hardware acceleration can reduce the need for exact fine-grained VSD, but at a very minimum, coarse-grained VSD will always be necessary to handle increasingly complex 3D scenes.

It should be noted that VSD techniques, within the context of interactive 3D applications, are related to but not the same as world partitioning techniques. A *world partitioning* technique is one which attempts to reduce the amount of the world (i.e., the number of 3D objects) which must be examined to update the environment from frame to frame. In an interactive 3D environment, the environment may change; objects might move around, change their state, and so forth. VSD techniques, if seen purely from the viewpoint of the visibility problem, are not the same as world partitioning techniques, because changes in the environment might take place in a portion of the world that is currently invisible. VSD techniques reduce the amount of work which needs to be done drawing the scene, but similar techniques are needed to reduce the amount of work updating the entire dynamic 3D environment. Typically, world partitioning techniques are an extension of a VSD technique. This means that we determine which parts of the world are visible, then allow objects in or "near" the visible area to change or move around. The rest of the environment is "frozen in time" and does not change until it becomes visible or nearly visible. The rationale is that areas near the current viewpoint will likely become visible in a few frames if the viewpoint moves in a continuous manner through the environment (as opposed to teleporting around randomly), so nearby areas are updated even if they are not yet completely visible. Ideally, of course, we would allow the entire world—all objects—to update each frame, but with larger worlds this is impossible, given that RAM and processing power are not infinite. Therefore, the compromise usually taken is to couple a VSD algorithm with a world partitioning algorithm. As with VSD, a primary concern of world partitioning schemes is avoiding sequentially looping through and testing many objects or areas only to find out that they are currently inactive; an efficient scheme computes the active regions near the viewer without exhaustive search. We see examples of this in Chapter 5.

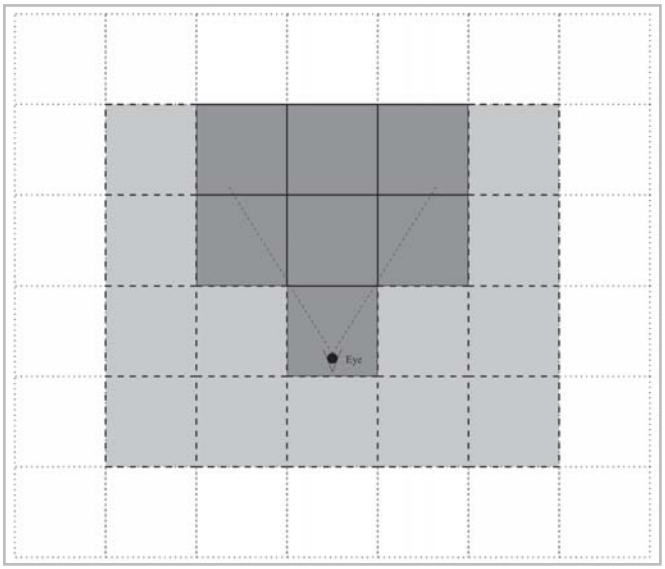


Figure 4-3: World partitioning and VSD schemes. If we partition the 3D world into a series of cubical areas, then only those areas within the field of view (solid lines) are visible. However, all visible areas and some additional "almost visible" areas (dashed lines) can be made active, so that the part of the world around the viewer updates. Areas too far away from the viewer (dotted lines) are inactive and not updated. Since active areas depend on the viewer location, they also change whenever the viewer moves. An efficient scheme will not need to perform an exhaustive search of all areas to determine which are currently active.

With an understanding of the correctness and efficiency issues that VSD algorithms address, and their relationship to world partitioning schemes, let's now look at a number of VSD techniques.

Back-Face Culling

One of the simplest VSD algorithms is the back-face culling algorithm. This algorithm mainly addresses the efficiency issue of VSD. In 3D graphics, to *cull* an object means to remove it from further consideration. Thus, back-face culling refers to the removal from further consideration of faces (polygons) which are back-facing. Let's explore this idea more closely.

Principally, we use polygons to model the surface of our 3D objects. If all of our objects are closed so that the interior may never be seen, this implies that we only ever see the outward-facing surface of the object. This means that we cannot see any part of the surfaces (polygons) pointing away from the viewer; the back side of the surface points toward the interior of the object, which we never see because it is always obstructed by the outward-facing polygons, and the front side of the surface points away from the viewer, meaning it also cannot be seen. Put simply, this means that any polygons facing away from the viewer—called *back-facing polygons*—need not be drawn.

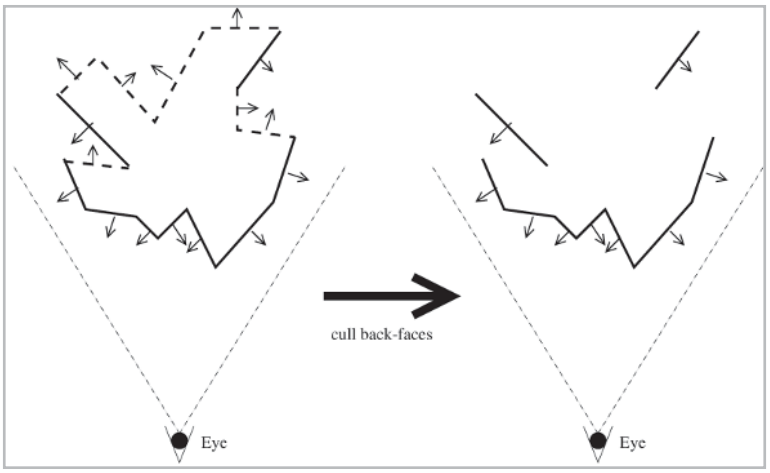


Figure 4-4: Back-facing polygons (dashed lines) can never be seen by the camera and are not drawn.

Not drawing back-facing polygons means, however, that clipping away parts of the front-facing polygons—against the near *z* clip plane, for instance—allows us to see straight through the object, since none of its back-faces are drawn. You can notice this behavior in almost every polygon-based 3D game, typically with the enemies. If you come too close to a polygonal mesh, it will be clipped against the near-z plane, and you see through its polygonal hollow shell. This is typically avoided by implementing a collision detection algorithm (see Chapter 8) so that the viewer cannot move so close to an object that it is clipped against the near *z* plane, but due to inaccuracy with the collision detection sometimes the viewer can get too close to an object.

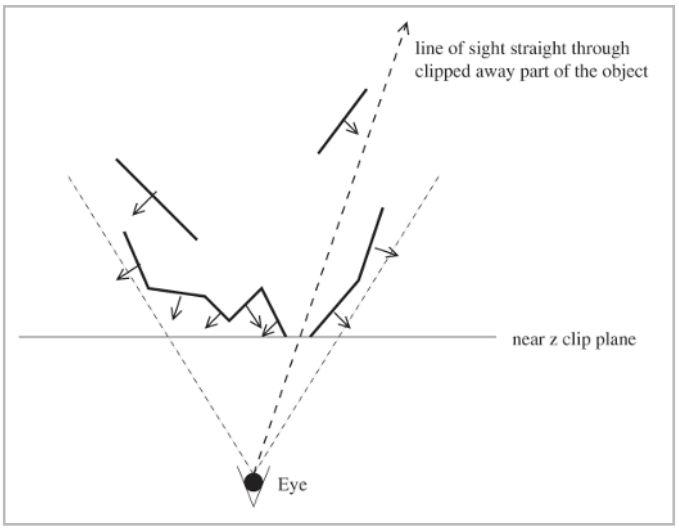


Figure 4-5: Clipping away parts of near polygons means that we see through the back side of the object, since the back-facing polygons are not drawn at all.

Determination of back-faces and front-faces is a straightforward matter. In camera space, we simply examine the sign of the *z* coordinate of the surface normal vector. In world space, we can take the dot product of the surface normal vector with the vector from the polygon's center to the

camera; if the dot product is negative, then the polygon is back-facing. The back-face culling algorithm is simple: if the polygon is back-facing, do not send it to the rasterizer to be drawn.

3D Convexity and Back-Face Culling

We encountered the concept of convexity in 2D when discussing the clipping and rasterization 2D polygons. As we have seen, the convexity of a polygon in 2D allows for a number of simplifications on various polygonal operations. Rasterization in horizontal spans and simplified analytical clipping are two operations we have seen that rely on the convexity of the 2D polygons on which they operate.

Convexity also plays a role in 3D. A polygonal 3D object is convex if it has no "dents" or "holes" in it. More formally, for a convex object, a line segment drawn between any two points on the object lies entirely within the object. 3D convexity means that the entire object is approximately sphere-shaped, just as 2D convex polygons were approximately O-shaped.

If the only object being viewed is a single convex polygonal 3D mesh, then back-face culling is the only VSD algorithm needed to correctly display the mesh. With a convex mesh, no front-facing polygon can obscure any part of any other front-facing polygon; this could only occur if a "dent" in the polyhedron caused some front-facing polygons to jut in front of other front-facing polygons. Since no front-facing polygons can obscure any others, and since we only see the front-facing polygons, this means we can simply draw all front-facing polygons in any order, and achieve a correct display—if our mesh is convex.

In and of itself, this observation is interesting, but not immediately useful. Typically the 3D objects we deal with are not convex. For instance, none of the objects we created with Blender in Chapter 3 (with the exception of the cube) were convex. Furthermore, scenes consist of many objects, not just one object. Lack of convexity implies that we need to resort to another VSD scheme to produce a correct display. For instance, until now we have use the painter's algorithm, which we look at again in this chapter, to draw the polygons in a back to front order. The *z* buffer algorithm, also covered later in this chapter, is another way of producing a correct display in the absence of convexity. These algorithms incur a computational expense for resolving the visibility among front-facing polygons by performing depth comparisons or computations.

Other VSD techniques, instead of attempting to compensate for the lack of convexity, try to create and leverage convexity. Two schemes that we look at later, the BSP tree and portals, rely on a convex partitioning of geometry. In this way, the convexity observation again becomes useful. By partitioning our objects into disjointed convex pieces, the VSD problem is raised to a higher level of ensuring correct visibility among the convex pieces; within each convex piece, we simply use back-face culling and perform no further VSD within the convex piece.

We return to this topic of convexity in more detail when we have looked at the BSP and portal schemes.

Sample Program: backface

The sample program `backface` implements back-face culling. Its structure is simple and typical of an l3d program. We derive the custom world class for program `backface` from the new class

`l3d_world_backface`. If you have read the introductory companion book *Linux 3D Graphics Programming*, you can see that this program is nearly identical to the `fltsim` program.

Run the program and notice the display in the upper left of the screen, showing the number of polygons sent to the rasterizer. Only the front-facing polygons are drawn, as you can see by the drawn polygon count.

Listing 4-1: `shapes.h`, shape definition header file for program `backface`

```
#include "../lib/geom/object/object3d.h"

class pyramid:public l3d_object {
    int xtheta;
  public:
    pyramid(void);
    virtual ~pyramid(void);
    int update(void);
};
```

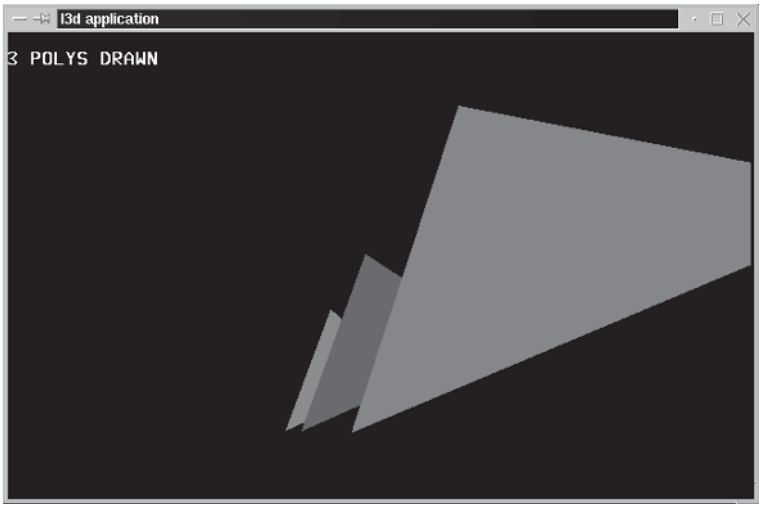


Figure 4-6: Output from sample program `backface`, illustrating the number of polygons drawn.

Listing 4-2: `shapes.cc`, shape definition code for program `backface`

```
#include "shapes.h"

#include <stdlib.h>
#include <string.h>

pyramid::pyramid(void) :
    l3d_object(4)
{
  (*vertices)[0].original.set
  (float_to_l3d_real(0.),
   float_to_l3d_real(0.),
   float_to_l3d_real(0.),
   float_to_l3d_real(1.));
  (*vertices)[1].original.set
  (float_to_l3d_real(10.0),
   float_to_l3d_real(0.),
   float_to_l3d_real(0.),
```

```
 float_to_l3d_real(1.));
(*vertices)[2].original.set
(float_to_l3d_real(0.),
 float_to_l3d_real(10.),
 float_to_l3d_real(0.),
 float_to_l3d_real(1.));
(*vertices)[3].original.set
(float_to_l3d_real(0.),
 float_to_l3d_real(0.),
 float_to_l3d_real(10.),
 float_to_l3d_real(1.));

int pi;
pi = polygons.next_index();
polygons[pi] = new l3d_polygon_3d_flatshaded(3);
printf("before: %p", polygons[pi]->vlist);
polygons[pi]->vlist = &vertices;
printf("after: %p", polygons[pi]->vlist);
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=0;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=1;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=3;
polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();

pi = polygons.next_index();
polygons[pi] = new l3d_polygon_3d_flatshaded(3);
polygons[pi]->vlist = &vertices;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=2;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=3;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=1;
polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();

pi = polygons.next_index();
polygons[pi] = new l3d_polygon_3d_flatshaded(3);
polygons[pi]->vlist = &vertices;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=0;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=2;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=1;
polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();

pi = polygons.next_index();
polygons[pi] = new l3d_polygon_3d_flatshaded(3);
polygons[pi]->vlist = &vertices;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=3;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=2;
(*polygons[pi]->ivertices)[polygons[pi]->ivertices->next_index()].ivertex=0;
polygons[pi]->compute_center();polygons[pi]->compute_sfcnormal();

num_xforms = 2;
xtheta=0;
modeling_xforms[0] = l3d_mat_rotx(xtheta);
modeling_xforms[1].set
( float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.),
  float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.), float_to_l3d_real(0.),
  float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.), float_to_l3d_real(0.),
  float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(0.), float_to_l3d_real(1.) );

modeling_xform=
  modeling_xforms[1] | modeling_xforms[0];
}
```

```
pyramid::~pyramid(void) {
  for(register int i=0; i<polygons.num_items; i++) {delete polygons[i]; }
}

int pyramid::update(void) {
  xtheta += 10; if (xtheta>359) xtheta-=360;
  modeling_xforms[0]=l3d_mat_rotx(xtheta);
  modeling_xform=
    modeling_xforms[1] | modeling_xforms[0];
}
```

Listing 4-3: `main.cc`, main program file for program `backface`

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../lib/geom/object/object3d.h"
#include "../lib/geom/polygon/p3_flat.h"
#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rast3.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/geom/world/world.h"
#include "../lib/geom/world/w_back.h"
#include "../lib/system/fact0_2.h"
#include "../lib/pipeline/pi_wor.h"
#include "../lib/dynamics/plugins/plugenv.h"

#include "shapes.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdarg.h>

class my_world:public l3d_world_backface {
  public:
    my_world(void);
};

my_world::my_world(void)
    : l3d_world_backface(640,400)
{
  l3d_screen_info *si = screen->sinfo;

  camera->VRP.set(0,0,-50,0);
  camera->near_z = float_to_l3d_real(5.5);
  camera->far_z = int_to_l3d_real(500);

  int i,j,k,onum=0;

  i=10; j=0; k=20;
  k=0;

  //- create some pyramid objects
  for(i=1; i<200; i+=20) {
    objects[onum=objects.next_index()] = new pyramid();
```

```
    l3d_screen_info_indexed *si_idx;
    l3d_screen_info_rgb *si_rgb;

    l3d_polygon_3d_flatshaded *p;
    for(int pnum=0; pnum<objects[onum]->polygons.num_items; pnum++) {
      p = dynamic_cast<l3d_polygon_3d_flatshaded *>(objects[onum]->polygons[pnum]);
      if(p) {
        p->final_color = si->ext_to_native
                         (rand()%si->ext_max_red,
                          rand()%si->ext_max_green,
                          rand()%si->ext_max_blue);
      }
    }

    if (objects[onum]==NULL) exit;
    objects[onum]->modeling_xforms[1].set
    ( int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(i),
      int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(0), int_to_l3d_real(1),
      int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1), int_to_l3d_real(1),
      int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(0), int_to_l3d_real(1) );
    objects[onum]->modeling_xform =
      objects[onum]->modeling_xforms[1] |
      objects[onum]->modeling_xforms[0] ;
  }

  //- create a plugin object

  objects[onum=objects.next_index()] = new l3d_object(10);
  //- max 10 fixed vertices, can be overridden by plug-in if desired
  //- by redefining the vertex list

  objects[onum]->plugin_loader =
    factory_manager_v_0_2.plugin_loader_factory->create();
  objects[onum]->plugin_loader->load("../lib/dynamics/plugins/pyramid/pyramid.so");
  objects[onum]->plugin_constructor =
    (void (*)(l3d_object *, void *))
    objects[onum]->plugin_loader->find_symbol("constructor");
  objects[onum]->plugin_update =
    (void (*)(l3d_object *))
    objects[onum]->plugin_loader->find_symbol("update");
  objects[onum]->plugin_destructor =
    (void (*)(l3d_object *))
    objects[onum]->plugin_loader->find_symbol("destructor");
  objects[onum]->plugin_copy_data =
    (void (*)(const l3d_object *, l3d_object *))
    objects[onum]->plugin_loader->find_symbol("copy_data");

  l3d_plugin_environment *e = new l3d_plugin_environment
                              (NULL, screen->sinfo, NULL, (void *)"");

  if(objects[onum]->plugin_constructor) {
    (*objects[onum]->plugin_constructor) (objects[onum],e);
  }

  screen->refresh_palette();
}

main() {
  l3d_dispatcher *d;
  l3d_pipeline_world *p;
```

```
my_world *w;

factory_manager_v_0_2.choose_factories();
d = factory_manager_v_0_2.dispatcher_factory->create();

w = new my_world();
p  = new l3d_pipeline_world(w);
d->pipeline = p;
d->event_source = w->screen;

d->start();

delete d;
delete p;
delete w;
```

Class l3d_World_Backface

The new class `l3d_world_backface` has modified the main polygon processing loop, by overriding method `draw_all`, to perform back-face culling. Recall that we group polygons into objects, but can also access the polygons as individual polygons, through the `nonculled_polygon_nodes` linked list member of `l3d_object`. Until now, the only time we removed polygons from this list was if they were completely clipped away against the near *z* plane. Now, with back-face culling, we also remove any polygons from the list if they are back-facing. Only the polygons surviving all culling operations are inserted into the final polygon list to be drawn for each frame.

The following code snippet from class `l3d_world_backface` performs the back-face culling.

```
    //- vector from facet center to camera (at 0,0,0), for backface culling
    l3d_vector facet_to_cam_vec;
    //- facet's sfc normal relative to origin (aFacet.sfc_normal is relative
    //- to facet center)
    l3d_vector N;
    n = objects[iObj]->nonculled_polygon_nodes;
    while(n) {
      N = n->polygon->sfcnormal.transformed - n->polygon->center.transformed;

      facet_to_cam_vec.set (0 - n->polygon->center.transformed.X_,
                            0 - n->polygon->center.transformed.Y_,
                            0 - n->polygon->center.transformed.Z_,
                            int_to_l3d_real(0));

      //- dot product explicitly formulated so no fixed point overflow
      //- (facet_to_cam_vec is very large)
      //- (use of the inline function "dot" uses default fixed precision)
#include "../../math/fix_lowp.h"
#define BACKFACE_EPS float_to_l3d_real(0.1)
      if(( l3d_mulrr(FX_CVT(N.X_), FX_CVT(facet_to_cam_vec.X_)) +
           l3d_mulrr(FX_CVT(N.Y_), FX_CVT(facet_to_cam_vec.Y_)) +
           l3d_mulrr(FX_CVT(N.Z_), FX_CVT(facet_to_cam_vec.Z_)) > BACKFACE_EPS)
         //- equivalent to: if(N.dot(facet_to_cam_vec) > BACKFACE_EPS )
#include "../../math/fix_prec.h"

          &&
```

TEAM-FLY

```
      n->polygon->clip_near_z(camera->near_z))

  {
    //- polygon is not culled
  }else {

    //- take this polygon out of the list of nonculled_polygon_nodes
```

First, we compute the vector going from the center of the polygon to the camera. Since this computation takes place in camera coordinates, the camera is at location (0,0,0). Then, we take the dot product of the polygon's surface normal vector and the computed polygon-to-camera vector, using the sign to determine the front-face or back-face orientation of the polygon. Note the explicit formulation of the dot product as a sum of products instead of using the operator `dot` to control the precision of the computation (how many bits are used for fractional and whole parts) in the case where fixed-point math is being used, since otherwise the dot product computation could overflow and incorrectly compute the orientation of polygons with fixed-point math. Also note the use of an epsilon parameter to further allow for some inaccuracy in the computation; polygons must be front-facing by an amount larger than the epsilon parameter to be accepted as front-facing. (The "amount" that a polygon is front-facing is the scalar returned by the dot product computation, which is the cosine of the angle between the vectors.) Since we are performing the back-face computation in camera coordinates, it would actually suffice just to check the sign of the *z* component of the normal vector. This example, however, illustrates the more general dot product method of back-face computation, which could also be performed in world coordinates before expending the computational effort on transforming vertices into camera space, only to find out that the transformed vertices are not needed because they belong to back-facing polygons.

Class `l3d_world_backface` also displays a count of polygons that it sends to the rasterizer. Notice that for identical scenes, this number is lower for program `backface` than for program `fltsim`; the back-face culling reduces the number of polygons sent to the rasterizer.

Listing 4-4: `w_back.h`, declaration for the new world class performing back-face culling

```
#ifndef __W_BACK_H
#define __W_BACK_H
#include "../../tool_os/memman.h"

#include "../../tool_os/dispatch.h"
#include "../../view/camera.h"
#include "../../tool_2d/screen.h"
#include "../object/object3d.h"
#include "world.h"

class l3d_world_backface : public l3d_world
{

  public:
    l3d_world_backface(int xsize, int ysize) :
    l3d_world(xsize,ysize) {};
    virtual ~l3d_world_backface(void);

    /* virtual */ void draw_all(void);
};

#endif
```

Listing 4-5: w_back.cc, code for the new world class performing back-face culling

```
#include "w_back.h"
#include <stdlib.h>
#include <string.h>

#include "../object/object3d.h"
#include "../polygon/polygon.h"
#include "../../tool_2d/screen.h"
#include "../../tool_os/dispatch.h"
#include "../../raster/rasteriz.h"
#include "../../tool_2d/scrinfo.h"
#include "../../system/fact0_2.h"
#include "../../tool_os/memman.h"

l3d_world_backface::~l3d_world_backface() {
}

void l3d_world_backface::draw_all(void) {
  int iObj, iFacet;

  l3d_polygon_3d *plist[MAX_VISIBLE_FACETS]; int pnum=0;

  rasterizer->clear_buffer();

  for(iObj=0; iObj<objects.num_items; iObj++) {

    objects[iObj]->reset();

    if (objects[iObj]->num_xforms) {
      objects[iObj]->transform(objects[iObj]->modeling_xform);
    }

    objects[iObj]->camera_transform(camera->viewing_xform);

    l3d_polygon_3d_node *n;

    //- vector from facet center to camera (at 0,0,0), for backface culling
    l3d_vector facet_to_cam_vec;
    //- facet's sfc normal relative to origin (aFacet.sfc_normal is relative
    //- to facet center)
    l3d_vector N;
    n = objects[iObj]->nonculled_polygon_nodes;
    while(n) {
      N = n->polygon->sfcnormal.transformed - n->polygon->center.transformed;

      facet_to_cam_vec.set (0 - n->polygon->center.transformed.X_,
                            0 - n->polygon->center.transformed.Y_,
                            0 - n->polygon->center.transformed.Z_,
                            int_to_l3d_real(0));

      //- dot product explicitly formulated so no fixed point overflow
      //- (facet_to_cam_vec is very large)
      //- (use of the inline function "dot" uses default fixed precision)
#include "../../math/fix_lowp.h"
#define BACKFACE_EPS float_to_l3d_real(0.1)
      if(( l3d_mulrr(FX_CVT(N.X_), FX_CVT(facet_to_cam_vec.X_)) +
           l3d_mulrr(FX_CVT(N.Y_), FX_CVT(facet_to_cam_vec.Y_)) +
           l3d_mulrr(FX_CVT(N.Z_), FX_CVT(facet_to_cam_vec.Z_)) > BACKFACE_EPS)
         //- equivalent to: if(N.dot(facet_to_cam_vec) > BACKFACE_EPS )
#include "../../math/fix_prec.h"
```

```
          &&

          n->polygon->clip_near_z(camera->near_z))

      {
        //- polygon is not culled
      }else {

        //- take this polygon out of the list of nonculled_polygon_nodes
        if(n->prev) {
          n->prev->next = n->next;
        }else {
          objects[iObj]->nonculled_polygon_nodes = n->next;
        }

        if(n->next) {
          n->next->prev = n->prev;
        }
      }

      n = n->next;
    }

    objects[iObj]->apply_perspective_projection(*camera,
        screen->xsize, screen->ysize);

    n = objects[iObj]->nonculled_polygon_nodes;

    while(n) {
      if ( n->polygon->clip_to_polygon_2d(screen->view_win)) {

        plist[pnum++] = n->polygon;

        /* Use the average of all z-values for depth sorting */
        l3d_real zsum=int_to_l3d_real(0);
        register int ivtx;

        for(ivtx=0; ivtx<n->polygon->ivertices->num_items; ivtx++) {
          zsum += (**(n->polygon->vlist))[ (*n->polygon->ivertices)[ivtx].ivertex
                    ].transformed_intermediates[1].Z_ ;
        }
        n->polygon->zvisible = l3d_divri(zsum, ivtx);
      }
      n = n->next;
    }
  }

  /* Sort the visible facets in decreasing order of z-depth */
  qsort(plist, pnum, sizeof(l3d_polygon_3d *), compare_facet_zdepth);

  /* Display the facets in the sorted z-depth order (painter's algorithm) */

  for(int i=0; i<pnum; i++) {
    plist[i]->draw(rasterizer);
  }

  char msg[256];
  sprintf(msg,"%d POLYS DRAWN", pnum);
  rasterizer->draw_text(0,16,msg);
}
```

Although it is not needed very often, if you really want to see both sides of a surface, you either need to use two polygons to model the front and back sides of the surface, or you need to skip the back-face culling operation. Note that if you skip the back-face culling operation, and want back-facing polygons to be drawn, you need to make a change to the rasterization routines in l3d, which all expect polygon vertices to be in a particular order, namely clockwise. Back-facing polygons have a reverse orientation, and thus have their vertices in a counterclockwise order, which would need to be handled within the rasterizer. In a sense, the fact that the rasterizer does not draw polygons with a reversed orientation is also a form of back-face culling (and is in fact the screen-space method which OpenGL uses), but it can be faster to eliminate sending the polygon to the rasterizer at all by doing the back-face determination using the surface normal vector.

View Frustum Culling

View frustum culling is another VSD algorithm which, like back-face culling, focuses on the efficiency side of VSD. With view frustum culling, we cull polygons or groups of polygons which are completely outside of the *view frustum*. The view frustum, also called the *view volume*, is the area in 3D space visible to the virtual viewer. The viewer does not see everything in the world; the horizontal and vertical field of view terms, which involve the size of the display window, limit the amount of the world which can be seen. Nothing outside of the window boundaries can be seen. Furthermore, neither objects closer than the near z plane nor farther than the far z plane can be seen. Geometrically, then, the view volume is a truncated pyramid with its apex at the camera. Objects inside the view volume are visible; objects outside the view volume are invisible.

The idea behind view frustum culling is to mathematically represent the view frustum in 3D space, then to test objects to see if they are in the frustum or not. If they are, they are processed further. If not, they are discarded from further processing immediately. Clearly, the earlier we can perform the view volume containment test, the more work we can potentially save.

Defining a View Frustum

The view frustum can be represented by six planes. The frustum itself is then the area inside of the intersection of all six planes.

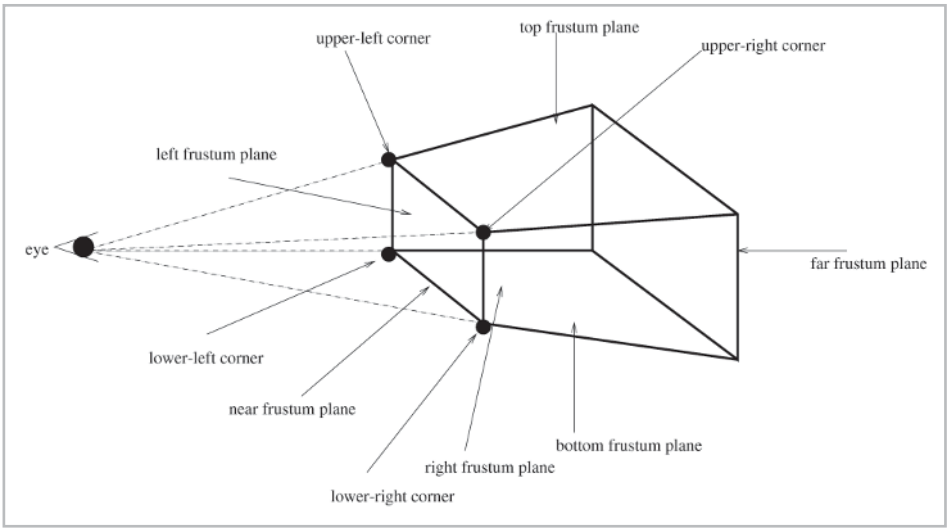


Figure 4-7: The view frustum can be defined by six planes: near, far, left, right, top, and bottom. The corner points are computed via reverse projection of the screen space corner points, and are used to compute the equations for the left, right, top, and bottom planes.

The near and far planes of the frustum are the near and far *z* clipping planes, each defined by a plane equation of the form $z=k$, where k is the distance of the plane (near or far) from the eye. Until now, we have not actually clipped against a far *z* plane, but the idea is the same as with the near *z* plane, with the exception that the far *z* plane's normal points in the opposite direction as the near *z* plane's, because we clip against the near side of the near plane, but against the far side of the far plane.

The remaining four planes of the frustum are a bit trickier to define. They represent the left, right, bottom, and top limits to the field of vision, due to the finite size of the projection window. Recall that knowing three points lying on a plane can define the plane; we only need to use the three points to form two unique vectors, take the cross product to find the plane normal (the A, B, and C terms of the plane equation), and then use one of the three points to solve for the D term of the plane equation. Method `l3d_plane::align_with_points` computes the plane coefficients based on three given points; remember that the order of the points passed to the method is important, because it determines the front/back orientation of the plane.

To define the remaining four planes of the frustum, we first calculate the camera space coordinates of the four corners on the near plane which, when projected from 3D to 2D, yield the four corners of the screen in screen space. In other words, we take a reverse projection of the four corners of the screen in 2D screen space, giving us four corner points in 3D camera space lying on the near *z* plane. Each view frustum plane passes through two of these points.

For instance, (0,0), (319,0), (319,239), (0,239) would be the screen space corner coordinates for a 320 240 window; we would reverse project these onto the near *z* plane using the reverse projection formulas from Chapter 2. Recall that reverse projection first finds the *z* value of the projected point by using the plane equation of the plane on which the 3D point must lie, then

multiplies the projected x and y by the computed 3D z value to obtain the 3D x and y values, thus reversing the perspective division by z. See Chapter 2 for the full equations, which involve the field of view terms and the screen size. Note that in this case we actually don't need to use the plane equation to compute the z value; since we are reverse projecting onto the near z plane, where z is a constant, we already know the z value to use in the reverse projection formulas of Chapter 2.

We then use consecutive pairs of these reverse projected corner points, along with the camera space coordinates of the camera itself (which is conveniently (0,0,0) in camera space), and calculate a plane given these three points, because these three points lie exactly on each side of the view frustum. Assuming we call the reverse projected corner points of the screen "upper-left," "upper-right," "lower-right," and "lower-left," the following list shows us which points to use to calculate a particular frustum plane.

- Top frustum plane: Camera location (0,0,0), upper-right, and upper-left.
- Right frustum plane: Camera location (0,0,0), lower-right, and upper-right.
- Bottom frustum plane: Camera location (0,0,0), lower-left, and lower-right.
- Left frustum plane: Camera location (0,0,0), top-left, and lower-left.



NOTE In the above list, the order of the points has been carefully chosen so as to define the inside side of the plane to be inside of the frustum; in other words, when viewed from the inside of the frustum, the specified points appear clockwise. This is important for the method `l3d_plane::align_with_points`, as mentioned earlier.

Computing the Frustum in World Coordinates

The above procedure gives us frustum planes, defined in terms of plane equation coefficients, in camera space. We can also perform the same procedure by using the world space locations by multiplying each point by the inverse camera transformation matrix just before using the point to compute the frustum plane. This gives us the frustum planes in world coordinates instead of camera coordinates. This can be useful to allow us to perform frustum culling in world space rather than camera space, which allows us to cull objects before transforming them to camera space. Note that such world space culling only saves computational effort if we use a simplified bounding volume to represent the geometry for the culling purposes; see the section titled "Bounding Spheres and the View Frustum."

For instance, to compute the top frustum plane in world coordinates, we multiply camera location (0,0,0) by the inverse camera transformation matrix to get the camera's position in world space. Similarly, we multiply the upper-right and upper-left corner points by the inverse camera matrix to obtain these points in world space instead of camera space. Then using these world space coordinates, we compute the plane equation.

Computing the near and far frustum planes in world space also requires us to create planes using three points in world space, not camera space. This is also straightforward: the near and far planes are defined, in camera space, by an equation of the form $z=k$, where k is a constant defining the distance from the eye to the plane. Therefore, in camera space, any point of the form (x,y,k), for arbitrary x and y, lies on the plane in question. We can choose any three points (x,y,k) that do not lie in a straight line to define the plane in camera space; then, we multiply these camera space points

by the inverse camera matrix to obtain world space points from which we can compute the world space plane. A convenient choice of non-collinear (not lying in a straight line) points for the near plane is (0,0,*k*), (1,0,*k*), and (0,1,*k*); for the far plane, use the same points in reverse order: (0,1,*k*), (1,0,*k*), (0,0,*k*). The far plane's normal must point in the opposite direction of the near plane's normal so that the inside side of each plane lies on the inside of the frustum—hence, the reversal of point ordering for near and far planes. For instance, if the near *z* plane is defined in camera space as z=1.5, we would take points (0,0,1.5), (1,0,1.5), and (0,1,1.5) as points known to be on the near plane in camera space. Then we multiply these points by the inverse camera matrix to obtain world space points. Finally, we compute the plane using these three points.

Class l3d_Viewing_Frustum

Class `l3d_viewing_frustum` represents a view frustum. It stores six planes for the six sides of the frustum, in member variables `top`, `bottom`, `left`, `right`, `near`, and `far`.

The method `create_from_points` creates a view frustum given four reverse projected corner points (as covered earlier), a near plane, a far plane, and a transformation matrix. The routine aligns the six planes of the frustum using the scheme described in the previous section: the top, bottom, left, and right planes are created using the eye point and two of the four corner points. The front and back planes are created using a simple non-collinear set of three points located on the near and far planes. Creation (or alignment, depending on how you view it) of all planes takes place through method `l3d_plane::align_with_points`. Before all points are passed to this method, they are multiplied with the transformation matrix `xform`, which is passed as a parameter. By default, the frustum creation takes place in camera coordinates and yields a set of planes representing the location of the frustum in camera coordinates. By passing a transformation matrix that changes from camera coordinates to world coordinates, we can compute the frustum in world coordinates, which is generally more desirable. This means that we pass the inverse viewing matrix from the camera as the `xform` parameter to obtain the frustum planes in world space. To obtain the frustum in camera space, pass an identity matrix as the `xform` parameter.

The method `intersect_with_sphere` tests to see if a sphere is inside of the view frustum. The next sections cover this method and the idea in more detail.

Listing 4-6: `vfrust.h`

```
#ifndef __VFRUST_H
#define __VFRUST_H
#include "../../tool_os/memman.h"
#include "../point/point.h"
#include "../plane/plane.h"
#include "../boundvol/boundsph.h"

class l3d_viewing_frustum {
  public:
    l3d_plane top, bottom, left, right, near, far;
    void create_from_points(const l3d_point & top_left,
                            const l3d_point & top_right,
                            const l3d_point & bot_right,
                            const l3d_point & bot_left,
                            const l3d_real  & near_z,
                            const l3d_real  & far_z,
                            const l3d_matrix& xform);
```

```
    int intersects_sphere(const l3d_bounding_sphere & sphere);
};

#endif
```

Listing 4-7: vfrust.cc

```
#include "vfrust.h"
#include "../../tool_os/memman.h"

void l3d_viewing_frustum::create_from_points
(const l3d_point & top_left,
 const l3d_point & top_right,
 const l3d_point & bot_right,
 const l3d_point & bot_left,
 const l3d_real  & near_z,
 const l3d_real  & far_z,
 const l3d_matrix& xform)
{
  l3d_point eye(0,0,0,1);

  top.align_with_points(xform|eye, xform|top_right, xform|top_left);
  bottom.align_with_points(xform|eye, xform|bot_left, xform|bot_right);
  left.align_with_points(xform|eye, xform|top_left, xform|bot_left);
  right.align_with_points(xform|eye, xform|bot_right, xform|top_right);

  l3d_point p1, p2, p3;

  p1.set(int_to_l3d_real(0), int_to_l3d_real(0), near_z, int_to_l3d_real(1));
  p2.set(int_to_l3d_real(1), int_to_l3d_real(0), near_z, int_to_l3d_real(1));
  p3.set(int_to_l3d_real(0), int_to_l3d_real(1), near_z, int_to_l3d_real(1));
  near.align_with_points(xform|p1,xform|p2,xform|p3);

  p1.set(int_to_l3d_real(0), int_to_l3d_real(1), far_z, int_to_l3d_real(1));
  p2.set(int_to_l3d_real(1), int_to_l3d_real(0), far_z, int_to_l3d_real(1));
  p3.set(int_to_l3d_real(0), int_to_l3d_real(0), far_z, int_to_l3d_real(1));
  far.align_with_points(xform|p1,xform|p2,xform|p3);
}

int l3d_viewing_frustum::intersects_sphere(const l3d_bounding_sphere & sphere)
{
  return
    (
      sphere.distance_to_plane(top) > -sphere.radius &&
      sphere.distance_to_plane(bottom) > -sphere.radius &&
      sphere.distance_to_plane(left) > -sphere.radius &&
      sphere.distance_to_plane(right) > -sphere.radius &&
      sphere.distance_to_plane(near) > -sphere.radius &&
      sphere.distance_to_plane(far) > -sphere.radius
    );
};
```

Using the Frustum Planes

Once we have the frustum planes, we then can test each polygon for containment within the frustum. A polygon is within the frustum if all of its vertices are in the frustum. A vertex is in the frustum if it is on the inside side of every frustum plane, which we can evaluate simply by evaluating the plane equation using the vertex. If a polygon is outside of the view frustum, it is removed from further processing by removing it from the `nonculled_polygon_nodes` list, just as we did for back-face culling.

Note that in reality, we usually do not perform frustum culling on single polygons, but instead on entire objects or even groups of objects. This is hierarchical frustum culling, our next topic.

Hierarchical View Frustum Culling

Individually checking every polygon for containment within the view frustum is very inefficient. We can be more efficient by introducing a hierarchy into our checks. Instead of checking each polygon, we can first perform a coarse check at the object level. If an entire object is found to be outside of the view frustum, then all of its polygons must also lie outside of the frustum, so we don't need to waste time checking every polygon in the object. If the object is entirely inside the view frustum, then all of its polygons are also inside. If the object is partially inside and partially outside the view frustum, then we must check each polygon individually.



NOTE With hardware acceleration, it might be faster to consider partially contained bounding volumes as being fully contained, and simply send all of their geometry to the hardware to be drawn. As we said at the beginning of this chapter, this may be inefficient, but hardware acceleration can often make the actual time penalty of such inefficiency virtually disappear.

Extending the hierarchy even further, we can collect objects into larger groups, and check each group first. If the entire group is inside or outside, then so are all of the constituent objects; if it is partially inside and partially outside, each constituent object must be checked separately. Checking the objects then proceeds as outlined above. The hierarchy can be extended to an arbitrary number of levels, starting with very large groups of objects, proceeding to medium-sized groups of objects, proceeding down to objects, proceeding down to individual polygons.

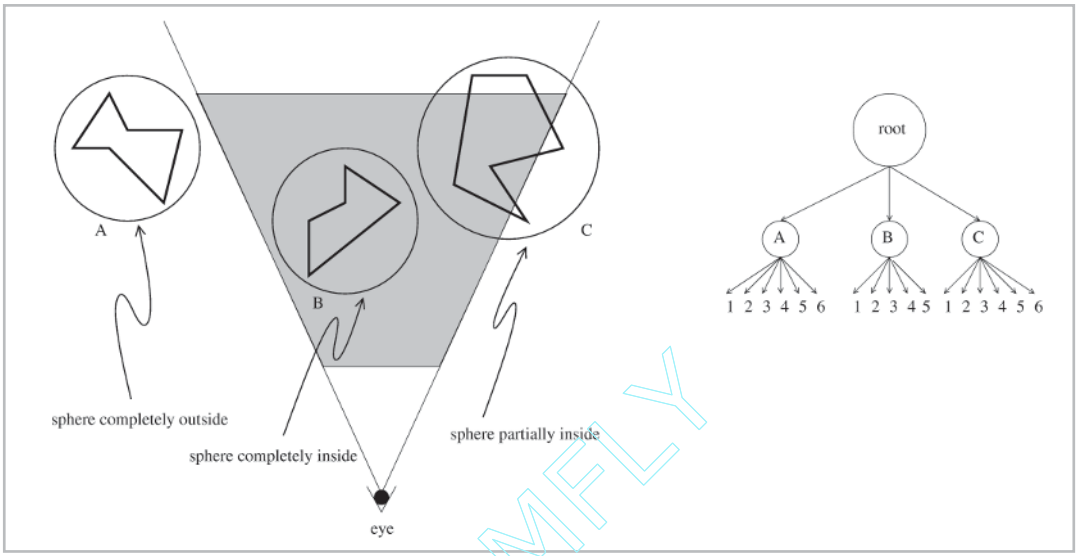


Figure 4-8: Hierarchical view frustum culling. First, check the hierarchically higher groups of polygons (A, B, and C), before investing time checking the individual polygons (1, 2, 3, 4, 5, 6, etc.). Here, A is completely outside, so all of its six polygons can be immediately skipped without traversing that branch of the hierarchy further. B is completely inside, so all of its polygons can be immediately passed on for further processing. C is partially inside and partially outside, so all of its individual polygons must be examined separately, thus requiring a further traversal of this branch of the hierarchy.

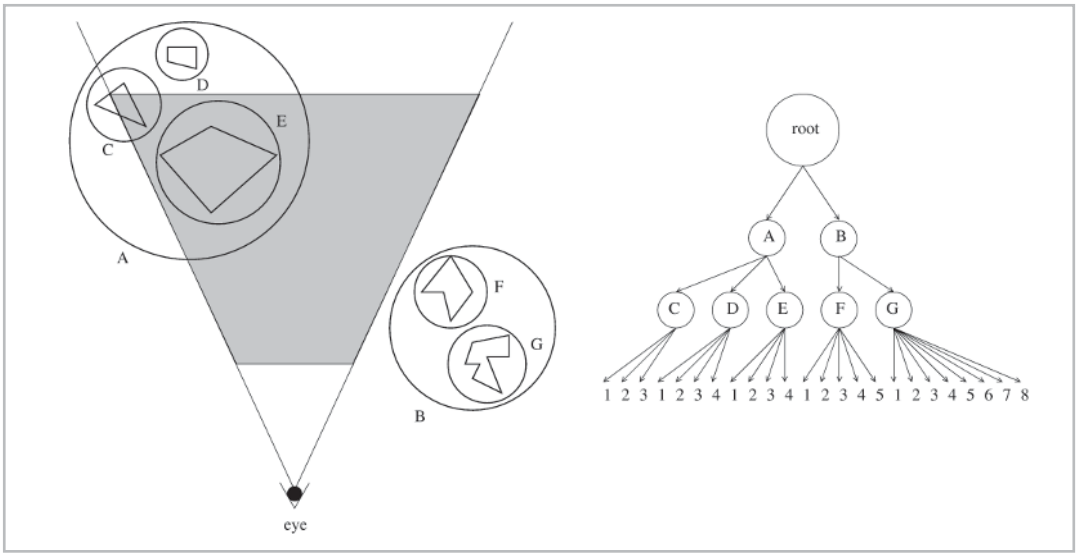


Figure 4-9: Multi-level hierarchy in view frustum culling. Here, group A is found to be partially inside the view frustum, so processing continues with all groups within A, namely, C, D, and E. Group C is partially inside, so processing continues with all polygons in group C. Group D is completely outside, so all of its polygons are immediately discarded. Group E is completely inside, so all of its polygons are immediately passed on for further processing. Group B, again at the top level of the hierarchy, is completely outside the frustum, meaning that everything within group B can be instantly discarded.

Team-Fly®

The key condition for such a hierarchical scheme to work is an efficient way of checking for view frustum containment of a group of polygons or objects. This group check must be faster than checking each individual polygon or object individually; otherwise, the whole scheme wastes time rather than saving it. The way that we can make group checks faster than individual checks is through the concept of a *bounding volume*. A bounding volume is a mathematically simple object, such as a sphere or a cube, that is large enough to contain all polygons of the object it bounds. Spheres are particularly simple and convenient for use as bounding volumes. Let's look more closely at how we can use spheres to save time in view frustum culling.

Bounding Spheres and the View Frustum

A 3D sphere can be defined in terms of a center and a radius. Let us assume that we have a sphere that is large enough to contain a polygonal model. (We look at computing such spheres later; it's quite easy.) We call such a sphere a *bounding sphere*. Checking for frustum containment of such a sphere is extremely fast and easy. It relies on the fact that evaluating a plane equation with a particular point (x,y,z) gives the shortest (i.e., perpendicular) distance between the plane and the point. Recall that the `side_of_point` function does exactly this: it evaluates the plane equation for a given point, and returns a positive, negative, or zero value if the point is in front of, behind, or on the plane, respectively. The function can be understood in terms of vector projections; the result of the `side_of_point` function is essentially the difference in the length of the projection onto the plane's normal vector of the vector from the origin to the point in question. In other words, the result of `l3d_plane::side_of_point` is the shortest distance from the point to the plane. If the point is on the plane, the distance from the point to the plane is zero. If the point is in front of the plane, it is a positive displacement from the plane; if it is behind the plane, it is a negative displacement from the plane.



CAUTION The plane's normal vector (the terms *A*, *B*, and *C* in the plane equation) must be normalized (have a length of 1) for the above computation to work. Plane vectors computed through the method `l3d_plane::align_with_points` satisfy this property, since this method creates a normalized vector. If you create your own planes by explicitly specifying plane equation coefficients, remember that the plane normal must be normalized. If it is not normalized, you can still use the plane equation to compute the distance between a point and the plane. To do this, first normalize the vector (divide it by its length). Then, evaluate the plane equation using the point for the *x*, *y*, and *z* values, using the normalized *A*, *B*, and *C* components of the normalized vector, and using the original *D* component of the plane equation. Generally, it's easiest to store normalized vectors from the beginning, so we don't have to re-normalize them later to do distance computations.

The trick that makes sphere checking easy is that the very definition of a sphere is based upon distance. A sphere is defined to be the set of all points in 3D space that are a certain distance, termed the radius, from a given center point. This means that the following are true:

- If the distance between a sphere's center and a plane is positive, negative, or zero, the center of the sphere is in front of, behind, or exactly on the plane, respectively.

- If the absolute value of the distance between a sphere's center and a plane is greater than the sphere's radius, the sphere is completely on one side of the plane; the sign of the distance determines which side it is on (inside or outside).
- If the absolute value of the distance between a sphere's center and a plane is less than the sphere's radius, the sphere is partially inside and partially outside the plane; in other words, the sphere intersects the plane.
- If the absolute value of the distance between a sphere's center and a plane is exactly equal to the sphere's radius, then the sphere grazes the plane.

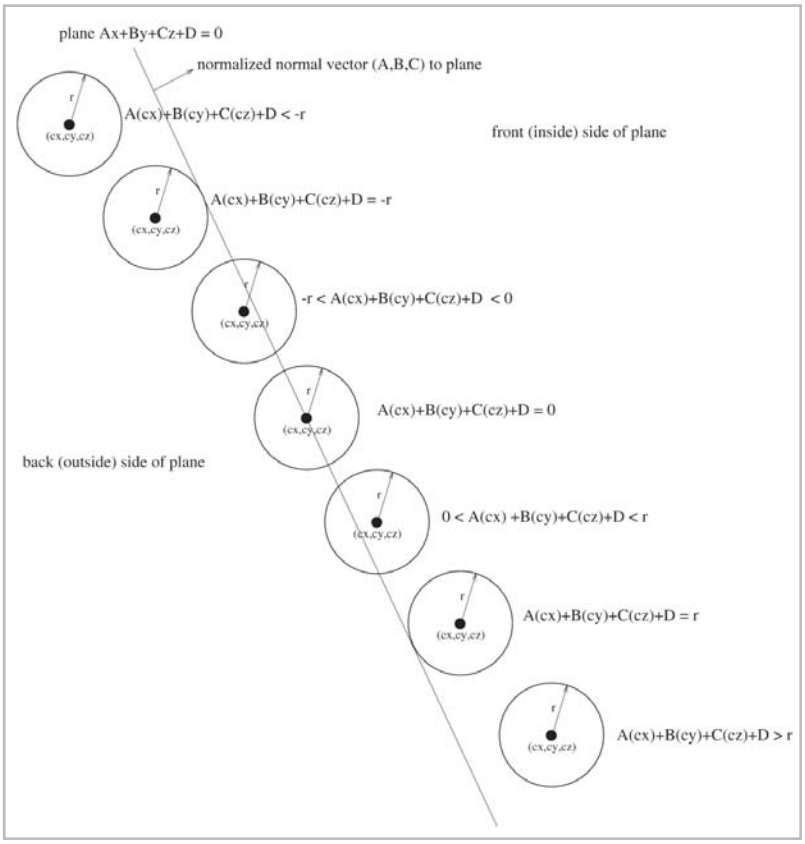


Figure 4-10: Checking for intersection between a bounding sphere and a plane. Given a plane with normalized normal vector (A,B,C), and a sphere with radius r and center at (cx,cy,cz), this diagram shows how to classify the position of the sphere with respect to the plane. We evaluate the plane equation with the center point of the sphere, and compare the result to r. The value of r tells us the position of the sphere with respect to the plane. From top to bottom, the sphere is completely outside of, grazing the outside of, mostly on the outside of, split exactly in half by, mostly inside of, grazing the inside of, or completely inside of the plane.

These facts mean that to check for containment of a bounding sphere in the view frustum, we evaluate the plane equation for each of the frustum's planes by using the center of the sphere as the (x,y,z) point in the plane equation. The result can be analyzed using the above facts to tell us if the sphere is completely outside, completely inside, or partially inside and partially outside of one individual frustum plane. If the sphere is completely outside of any one frustum plane, it and all of its bounded geometry are completely outside of the entire frustum, and can be eliminated from further processing. If the sphere is completely inside all frustum planes, then it and all of its bounded geometry are completely inside the entire frustum, and should be passed on for further

processing. In any other case, the sphere and its bounded geometry are partially inside and partially outside of the entire frustum, and view frustum culling must continue at the next more detailed level of the hierarchy for each polygon or object within the bounding sphere.

Note that this computation can be performed in either world space or camera space; we simply need the sphere's center point and the planes in the appropriate coordinate system.



TIP The classification of a bounding sphere with respect to a plane can also be interpreted as the detection of whether the sphere collides with the plane. See Chapter 8 for more on the topic of collision detection.

Also, notice that we can save time by first transforming only an object's bounding sphere from object space into world space to be tested against the world space frustum. If the sphere turns out to be outside of the frustum, we have saved the work of transforming all of the object's vertices. If the sphere turns out to be inside or partially inside the frustum, only then do we expend computational effort transforming the entire object geometry into world space.

Computing Bounding Spheres

We can compute a bounding sphere for a polygonal object by computing two values: a center and a radius.

Compute the center of the object by finding the maximum and minimum x, y, and z values among all vertices. Then, for each x, y, and z, add the maximum and minimum value and divide by two to arrive at the average middle position along the x, y, and z axes. This yields a single (x,y,z) location representing the center of the object. Then, with the center point, calculate the distance of each vertex from the center, using the 3D distance formula. Keep track of the maximum distance found. Then, define the radius of the sphere to be the maximum distance found. In this way, all vertices, and thus all polygons, are guaranteed to lie within the sphere, because they all have a distance smaller than the maximum distance from the sphere's center.

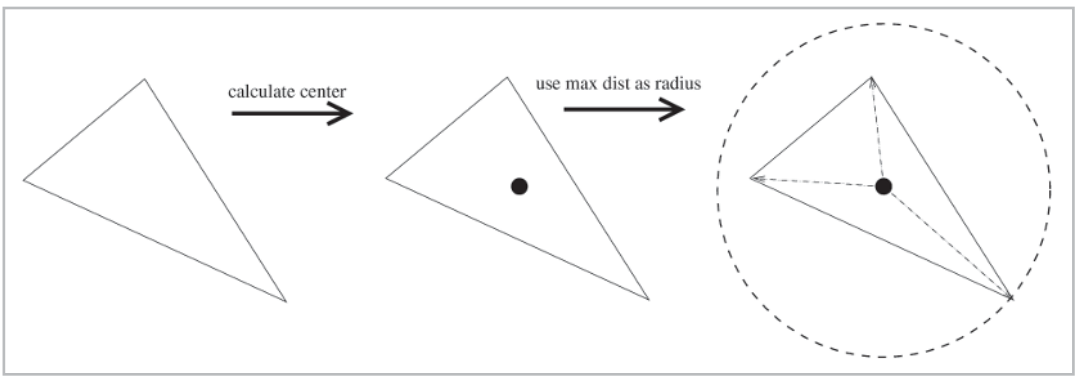


Figure 4-11: Computing a bounding sphere.

Note that regardless of how the object is rotated, the bounding sphere remains the same.



CAUTION If the object contains vertices that are not referenced by any polygon, then be sure not to use such vertices when computing the center and radius.

It is possible for an object to contain unused vertices—that is, vertices which are not referenced by any polygon. It is important not to use such vertices when computing the center or radius of the polygon, since such vertices do not contribute to the geometry of the object. The world editing scheme covered in Chapter 6 intentionally adds "unused" vertices to the polygonal mesh (vertices not referenced by any polygon) to encode non-geometric information within the mesh. This non-geometric information is a unique integer object identifier, which we can then use to associate arbitrary information with every polygonal mesh: a name, the name of a texture file to be associated with the mesh, and so forth. We should be careful that such non-geometrical vertices do not contribute to the bounding sphere computation.

Class l3d_bounding_sphere

Class `l3d_bounding_sphere` is the l3d class representing a bounding sphere.

Member variables `radius` and `center` store the radius and center of the bounding sphere. The center is stored as `l3d_coordinate`, so it has an original and a transformed location. The method `compute_around_object` computes the bounding sphere around a given object by computing the center and finding the maximum required radius among all of the object's vertices which are used by at least one polygon. Method `distance_to_plane` computes the distance between the center of the sphere and a plane, simply by calling the plane's `side_of_point` method with the center point; it is merely another interface to the same functionality.

The member variable `transform_stage`, along with methods `transform` and `reset`, are responsible for transforming the bounding sphere and tracking how many times the sphere has already been transformed, just as is the case with the vertices of objects. Since a bounding sphere is sort of a simplified version of an object, we also provide for similar means of transforming the bounding sphere.

Transforming a bounding sphere is as simple as transforming its center point; no transformation we use for the code in this book can change the shape or orientation of a sphere (only a non-uniform scaling or a shearing transformation could squash or tug the sphere into an asymmetric ellipsoid with an identifiable orientation). We track the number of times the sphere has been transformed because the bounding sphere should, eventually, be transformed the same number of times (and with the same matrices) as the geometry it bounds. We say "eventually" because in the case of a culling operation, the whole idea is to save time by first transforming only the bounding sphere, then later transforming the geometry if needed. However, for other operations (such as collision detection), we transform the geometry, then use the bounding sphere later, expecting that the bounding sphere has also been transformed along with the geometry. In such a case, the bounding sphere is transformed after the geometry, not before. We use the `transform_stage` variable to synchronize the transformations of the bounding sphere with those of its underlying geometry.

We saw earlier how to check for containment of a bounding sphere within the view frustum; we check the distance between the sphere's center with every plane in the frustum. If the sphere is on the inside side of every plane, it is in the frustum; otherwise, it is outside. The method

`l3d_view_frustum::intersects_sphere`, which we saw earlier, performs exactly this test, using the bounding sphere's `distance_to_plane` method. The `intersects_sphere` method considers any sphere that is even partially within the frustum to be inside; the distance tests check the distance against the negative radius of the sphere. If the signed distance between the sphere's center and the plane is greater than the negative radius, this means that at least some part of the sphere is on the inside side of the plane. This implies that the view frustum test with bounding spheres, as implemented here, is a *conservative* test. It may conservatively flag an object as being "inside" the frustum, even if only a tiny part of the bounding sphere is inside the frustum. Since a bounding sphere cannot perfectly fit its underlying geometry (unless the underlying geometry is also a sphere), the actual geometry might be completely outside the frustum even if our test says it might be inside. But no geometry inside the frustum is ever rejected with our test. This is a general characteristic of conservative tests: they overestimate the work that might need to be done, but never underestimate it.

Listing 4-8: boundsph.h

```
#ifndef __BOUNDSPH_H
#define __BOUNDSPH_H
#include "../../tool_os/memman.h"

#include "../../system/sys_dep.h"
#include "../plane/plane.h"
#include "../../math/matrix.h"
#include "../vertex/coord.h"
class l3d_object;

class l3d_bounding_sphere {
  public:
    int transform_stage;
    virtual void reset(void);
    virtual void transform(const l3d_matrix &xform);

    l3d_real radius;
    l3d_coordinate center;
    l3d_real distance_to_plane(const l3d_plane & plane) const;
    void compute_around_object(const l3d_object *object);
};

#endif
```

Listing 4-9: boundsph.cc

```
#include "boundsph.h"
#include "../plane/plane.h"
#include "../object/object3d.h"
#include "../../tool_os/memman.h"

void l3d_bounding_sphere::reset(void) {
  transform_stage = 0;
  center.reset();
}

void l3d_bounding_sphere::transform(const l3d_matrix &xform) {
  transform_stage++;
  center.transform(xform, transform_stage);
}
```

```
l3d_real l3d_bounding_sphere::distance_to_plane(const l3d_plane & plane) const
{
  //- ensure the size of the plane's normal vector is 1, otherwise this
  //- computation doesn't work

  return plane.side_of_point(center.transformed);
}

void l3d_bounding_sphere::compute_around_object(const l3d_object *object) {
  int i;
  center.original.set(0,0,0,1);

  l3d_point min, max;
  int first_vtx=1;

  for(i=0; i<object->vertices->num_fixed_items; i++) {
    int vtx_is_referenced=0;
    for(int pi=0; pi<object->polygons.num_items && !vtx_is_referenced;pi++) {
      for(int vi=0; vi<object->polygons[pi]->ivertices->num_items; vi++) {
        if((*object->polygons[pi]->ivertices)[vi].ivertex == i) {
          vtx_is_referenced = 1;
          break;
        }
      }
    }

    if(vtx_is_referenced) {
      if(first_vtx) {
        first_vtx = 0;
        min.X_ = (*object->vertices)[i].original.X_;
        max.X_ = (*object->vertices)[i].original.X_;
        min.Y_ = (*object->vertices)[i].original.Y_;
        max.Y_ = (*object->vertices)[i].original.Y_;
        min.Z_ = (*object->vertices)[i].original.Z_;
        max.Z_ = (*object->vertices)[i].original.Z_;
      }else {
        if ( (*object->vertices)[i].original.X_ < min.X_) {
          min.X_ = (*object->vertices)[i].original.X_;
        }
        if ( (*object->vertices)[i].original.X_ > max.X_) {
          max.X_ = (*object->vertices)[i].original.X_;
        }
        if ( (*object->vertices)[i].original.Y_ < min.Y_) {
          min.Y_ = (*object->vertices)[i].original.Y_;
        }
        if ( (*object->vertices)[i].original.Y_ > max.Y_) {
          max.Y_ = (*object->vertices)[i].original.Y_;
        }
        if ( (*object->vertices)[i].original.Z_ < min.Z_) {
          min.Z_ = (*object->vertices)[i].original.Z_;
        }
        if ( (*object->vertices)[i].original.Z_ > max.Z_) {
          max.Z_ = (*object->vertices)[i].original.Z_;
        }
      }
    }
  }

  center.original.X_ =
    l3d_divri(min.X_ + max.X_, 2);
```

```
  center.original.Y_ =
    l3d_divri(min.Y_ + max.Y_, 2);
  center.original.Z_ =
    l3d_divri(min.Z_ + max.Z_, 2);

  radius = int_to_l3d_real(0);
  for(i=0; i<object->vertices->num_fixed_items; i++) {
    int vtx_is_referenced=0;
    for(int pi=0; pi<object->polygons.num_items && !vtx_is_referenced;pi++) {
      for(int vi=0; vi<object->polygons[pi]->ivertices->num_items; vi++) {
        if((*object->polygons[pi]->ivertices)[vi].ivertex == i) {
          vtx_is_referenced = 1;
          break;
        }
      }
    }

    if(vtx_is_referenced) {

      l3d_real dist_squared;
      dist_squared = dot((*object->vertices)[i].original - center.original,
                         (*object->vertices)[i].original - center.original);

      if(dist_squared > radius) {
        radius = dist_squared;
      }
    }
  }
  //- the value computed in the above loop was the max squared distance,
  //- now take the square root to retrieve the actual distance
  radius = l3d_sqrt(radius);

  reset(); //- copy original into transformed
}
```

Other Bounding Volumes

Other bounding volumes are also used in 3D graphics for the purposes of faster testing and early rejection. Two common ones are the arbitrarily oriented bounding box (OBB) and the axially aligned bounding box (AABB).

An *arbitrarily oriented bounding box* is a cube-like box, with pairwise parallel sides which fully encompass the geometry it bounds. Like a cube, it can be defined with eight points. In general, a bounding box is stored along with the geometry it bounds, and is transformed the same way. For instance, if we rotate an object, we rotate its bounding box as well. This means that a bounding box can be arbitrarily oriented in space. On the one hand, this makes bounding boxes good bounding volumes, because they can tightly "fit" the underlying geometry (i.e., the bounding volume contains relatively little empty space around the bounded geometry), but on the other hand, the arbitrary orientation of the bounding box means that we have six arbitrarily oriented faces and eight arbitrarily located points. For instance, computing if two arbitrarily aligned bounding boxes intersect is not trivial; any one point of one box might pierce any face of the other box. For the purposes of view volume culling, we can check all corner points of the bounding box against all planes of the view frustum, then use the same logic we used for spheres for determining containment in the view frustum. One special case which must be handled is a huge bounding box larger

than the frustum itself; all eight corners lie outside the frustum, but part of the box does lie within the frustum.

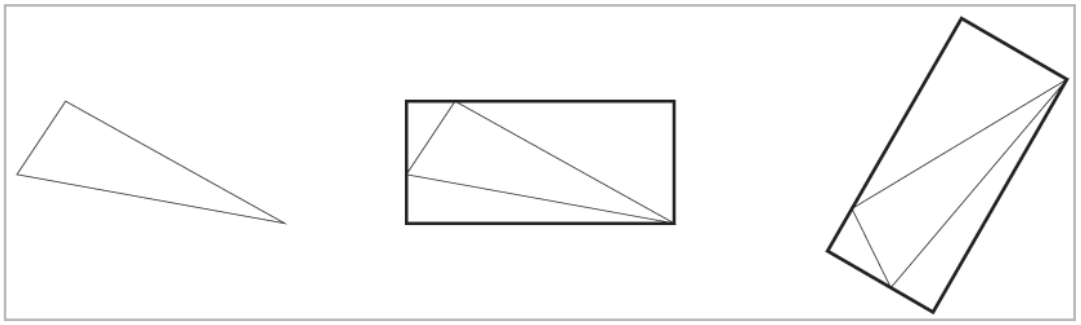


Figure 4-12: An arbitrarily oriented bounding box is transformed along with its geometry, making for a tight fit but possibly more complicated computations.

An *axially aligned bounding box* is the same as a bounding box, with the exception that all of its sides are always parallel to one of the *x*-*y*, *y*-*z*, or *x*-*z* planes. In other words, the normal vectors of the sides of the AABB must lie along one of the principal coordinate axes, hence the term "axially aligned." In contrast to an arbitrarily oriented bounding box, an axially aligned bounding box is not transformed along with its underlying geometry due to the alignment restriction on the sides. This means that as an object rotates, its AABB might need to be recomputed. Since the sides of an AABB are all conveniently aligned, we only need to store two opposite corner points of the AABB, not all eight corner points. AABBs generally provide a fairly poor fit to the underlying geometry, but certain operations on AABBs are very easy. For instance, intersection of two AABBs can be determined by three consecutive one-dimensional overlap tests; if the *x*, *y*, and *z* intervals defining the sides of the two AABBs overlap, then the AABBs themselves overlap; otherwise they are disjoint. For the purposes of view volume culling, AABBs can be handled like OBBs; we test the corner points of the AABB against the view frustum to see if the entire AABB is inside or outside of the view frustum.

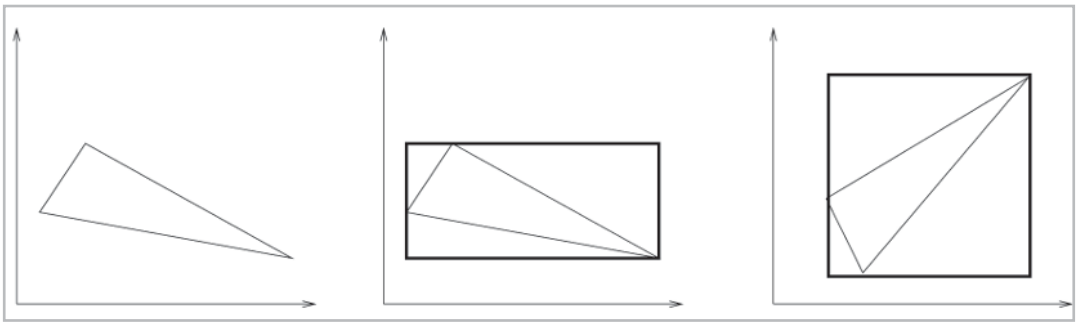


Figure 4-13: An axis-aligned bounding box has sides whose normal vectors lie along the principal coordinate axes. It provides a poor fit to geometry, but simplifies some computations.

Clipping Against the View Frustum

After culling objects against the view frustum, we know all objects whose bounding spheres lie at least partially within the frustum. We can then process these objects as normal—performing back-face culling, perspective projection, clipping to the 2D view window, and rasterization.

However, with the addition of the 3D view frustum, we now can make a change to one step: the clip against the 2D view window. Recall that projection of a polygon from 3D into 2D can lead to vertices which lie outside of the 2D view window. Our solution up to this point was to clip the projected polygons in 2D against the view window to ensure that only the part lying within the window would be drawn.

But clipping projected polygons in 2D to the view window has some quirks. In particular, the new vertices that may be generated from a 2D clipping operation do not directly result from a 3D to 2D projection of an original vertex in the polygon. We've mentioned this fact in a number of places before. This lack of original 3D information for the vertices generated by 2D clipping led us to develop a reverse projection strategy to obtain the corresponding 3D coordinate for such 2D vertices. We needed this for the Mesa/OpenGL texture mapping strategy of Chapter 2. Furthermore, clipping texture coordinates in 2D required us to clip u/z, v/z, and $1/z$, instead of u and v, because u/z, v/z, and $1/z$ can be linearly interpolated and clipped in 2D, whereas u and v cannot.

Reverse projection and the linearity of u/z, v/z, and $1/z$ in 2D screen space are important topics to understand, but we can sidestep these issues by performing clipping in 3D instead of in 2D. Instead of clipping projected polygons against the 2D view window, we can now clip world space polygons in 3D against the planes of the 3D view frustum. Mathematically, this produces exactly the same results as the clip in 2D to the 2D view window. In 2D, we clip against the edges of the view window; in 3D, we clip against the planes, computed via reverse projection, passing through these same edges and the eye. But by performing the clipping in 3D, we can directly clip the u and v values instead of u/z, v/z, and $1/z$. We saw this in the class `l3d_polygon_3d_textured`. Furthermore, this means that all polygon vertices in 2D directly result from the projection of a 3D vertex. This would eliminate the need for certain reverse projection operations that we used for the Mesa/OpenGL texture mapping strategy.

Personally, I feel that there are arguments for both 2D and 3D clipping, so you can choose whichever seems more comfortable to you. The next sample program eliminates 2D clipping entirely and does all clipping in 3D, so you can see how it is done.



CAUTION Do not confuse *culling* against the view frustum, which is a possibly hierarchical inside/outside classification operation, with *clipping* against the view frustum, which is an actual change of the underlying geometry of an object to ensure that only the part inside the frustum remains.

Sample Program: frustum

The sample program `frustum` performs view frustum culling on a number of tree objects which are loaded via a plug-in. The program displays on-screen how many objects have been culled via a bounding sphere test. Navigate around the scene, and notice how many objects are culled away when you are not looking at them.



Figure 4-14: Output from sample program `frustum`.

We load the objects in the main program file. After loading each object, we compute a bounding sphere around it. Then, the new world subclass `l3d_world_frustum`, which is covered in the next section, draws the world, incorporating view frustum culling.

Listing 4-10: `main.cc`, main program file for program `frustum`

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../lib/geom/object/obound.h"
#include "../lib/geom/polygon/p3_flat.h"
#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rast3.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/geom/world/w_vfrust.h"
#include "../lib/system/fact0_2.h"
#include "../lib/pipeline/pi_wor.h"
#include "../lib/dynamics/plugins/plugenv.h"
#include "../lib/geom/texture/tl_ppm.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdarg.h>

class my_world:public l3d_world_frustum {
  protected:
    l3d_texture_loader *texloader;
    l3d_surface_cache *scache;
  public:
    my_world(void);
    virtual ~my_world(void);
};

my_world::my_world(void)
    : l3d_world_frustum(320,240)
{
  l3d_screen_info *si = screen->sinfo;
```

TEAM-FLY

```
camera->VRP.set(0,0,-50,0);
camera->near_z = float_to_l3d_real(1.0);
camera->far_z = int_to_l3d_real(50);

//- for mesa rasterizer's reverse projection
rasterizer_3d_imp->fovx = &(camera->fovx);
rasterizer_3d_imp->fovy = &(camera->fovy);
rasterizer_3d_imp->screen_xsize = &(screen->xsize);
rasterizer_3d_imp->screen_ysize = &(screen->ysize);

int i,j,k,onum=0;

texloader = new l3d_texture_loader_ppm(si);
scache = new l3d_surface_cache(si);

for(i=0; i<150; i++) {

  //- create a plugin object
  l3d_object_clippable_boundable *oc;

  objects[onum=objects.next_index()] =
    oc =
      new l3d_object_clippable_boundable(10);
  //- max 10 fixed vertices, can be overridden by plug-in if desired
  //- by redefining the vertex list

  objects[onum]->plugin_loader =
    factory_manager_v_0_2.plugin_loader_factory->create();
  objects[onum]->plugin_loader->load("../lib/dynamics/plugins/vidmesh/vidmesh.so");
  objects[onum]->plugin_constructor =
    (void (*)(l3d_object *, void *))
    objects[onum]->plugin_loader->find_symbol("constructor");
  objects[onum]->plugin_update =
    (void (*)(l3d_object *))
    objects[onum]->plugin_loader->find_symbol("update");
  objects[onum]->plugin_destructor =
    (void (*)(l3d_object *))
    objects[onum]->plugin_loader->find_symbol("destructor");
  objects[onum]->plugin_copy_data =
    (void (*)(const l3d_object *, l3d_object *))
    objects[onum]->plugin_loader->find_symbol("copy_data");

  texloader = new l3d_texture_loader_ppm(si);
  scache = new l3d_surface_cache(si);

  char plugin_parms[4096];
  sprintf(plugin_parms, "%d %d %d  1 0 0 0 1 0 0 0 1 tree.obj tree.ppm tree.uv",
          rand()%200-100, 0, rand()%200-100);

  l3d_plugin_environment *e = new l3d_plugin_environment
                              (texloader, screen->sinfo, scache, plugin_parms);

  if(objects[onum]->plugin_constructor) {
    (*objects[onum]->plugin_constructor) (objects[onum],e);
  }

  oc->bounding_sphere.compute_around_object(oc);

}
```

```
  screen->refresh_palette();
}

my_world::~my_world(void) {
  delete texloader;
  delete scache;
}

main() {
  l3d_dispatcher *d;
  l3d_pipeline_world *p;
  my_world *w;

  factory_manager_v_0_2.choose_factories();
  d = factory_manager_v_0_2.dispatcher_factory->create();

  w = new my_world();
  p  = new l3d_pipeline_world(w);
  d->pipeline = p;
  d->event_source = w->screen;

  d->start();

  delete d;
  delete p;
  delete w;
}
```

Class l3d_World_Frustum

The new class `l3d_world_frustum` incorporates view frustum culling into the world's drawing behavior. It only overrides method `draw_all`.

Listing 4-11: `w_vfrust.h`

```
#ifndef __W_VFRUST_H
#define __W_VFRUST_H
#include "../../tool_os/memman.h"

#include "../../tool_os/dispatch.h"
#include "../../view/camera.h"
#include "../../tool_2d/screen.h"
#include "../object/object3d.h"
#include "world.h"

class l3d_world_frustum : public l3d_world
{

  public:
    l3d_world_frustum(int xsize, int ysize) :
    l3d_world(xsize,ysize) {};
    virtual ~l3d_world_frustum(void);

    /* virtual */ void draw_all(void);
};

#endif
```

Listing 4-12: w_vfrust.cc

```
#include "w_vfrust.h"
#include <stdlib.h>
#include <string.h>

#include "../object/obound.h"
#include "../polygon/polygon.h"
#include "../../tool_2d/screen.h"
#include "../../tool_os/dispatch.h"
#include "../../raster/rasteriz.h"
#include "../../tool_2d/scrinfo.h"
#include "../../system/fact0_2.h"
#include "../frustum/vfrust.h"
#include "../../tool_os/memman.h"

l3d_world_frustum::~l3d_world_frustum() {
}

void l3d_world_frustum::draw_all(void) {
  int iObj, iFacet;

  l3d_polygon_3d *plist[MAX_VISIBLE_FACETS]; int pnum=0;

  rasterizer->clear_buffer();

  l3d_viewing_frustum frustum;

  l3d_point top_left, top_right, bot_right, bot_left;
  const int XOFF=2,YOFF=2;

  top_left.set(l3d_mulrr(l3d_divrr(int_to_l3d_real(XOFF - screen->xsize/2),
                                   l3d_mulri(camera->fovx,screen->xsize)),
                         camera->near_z),
               l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->ysize/2 - YOFF),
                                   l3d_mulri(camera->fovy,screen->ysize)),
                         camera->near_z),
               int_to_l3d_real(camera->near_z),
               int_to_l3d_real(1));

  top_right.set(l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->xsize-XOFF - screen->xsize/2),
                                    l3d_mulri(camera->fovx,screen->xsize)),
                          camera->near_z),
                l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->ysize/2 - YOFF),
                                    l3d_mulri(camera->fovy,screen->ysize)),
                          camera->near_z),
                int_to_l3d_real(camera->near_z),
                int_to_l3d_real(1));

  bot_right.set(l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->xsize-XOFF - screen->xsize/2),
                                    l3d_mulri(camera->fovx,screen->xsize)),
                          camera->near_z),
                l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->ysize/2 - screen->ysize+YOFF),
                                    l3d_mulri(camera->fovy,screen->ysize)),
                          camera->near_z),
                camera->near_z,
                int_to_l3d_real(1));

  bot_left.set(l3d_mulrr(l3d_divrr(int_to_l3d_real(XOFF - screen->xsize/2),
                                   l3d_mulri(camera->fovx,screen->xsize)),
                         camera->near_z),
```

```
            l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->ysize/2 - screen->ysize+YOFF),
                                l3d_mulri(camera->fovy,screen->ysize)),
                      camera->near_z),
            camera->near_z,
            int_to_l3d_real(1));

frustum.create_from_points
(top_left, top_right, bot_right, bot_left,
 camera->near_z, camera->far_z,
 camera->inverse_viewing_xform);

int culled_obj_count;
culled_obj_count =0;
for(iObj=0; iObj<objects.num_items; iObj++) {

  objects[iObj]->reset(); //- also resets bounding sphere

  if (objects[iObj]->num_xforms) {

    l3d_object_clippable_boundable *obj;
    if((obj=dynamic_cast<l3d_object_clippable_boundable *> (objects[iObj])))
    {

      //- first, position *ONLY* the bounding sphere in world space
      obj->bounding_sphere.transform(objects[iObj]->modeling_xform);

      //- now, check if the bounding sphere is in the frustum
      if(! frustum.intersects_sphere(obj->bounding_sphere)) {
        culled_obj_count++;
        continue; //- frustum does not intersect sphere->discard object
      }else {
        //- sphere is at least partially in frustum. so, now,
        //- transform *ALL* of the object's vertices into world space.

        objects[iObj]->transform(objects[iObj]->modeling_xform);
      }
    }
    else {

      //- no bounding sphere available. transform all object vertices
      //- immediately.
      objects[iObj]->transform(objects[iObj]->modeling_xform);
    }
  }else {
    //- no object-to-world space xform applied, so obj coords
    //- are world coords; do frustum test on obj coords

    l3d_object_clippable_boundable *obj;
    if((obj=dynamic_cast<l3d_object_clippable_boundable *> (objects[iObj])))
    {
      if(! frustum.intersects_sphere(obj->bounding_sphere)) {
        culled_obj_count++;
        continue; //- frustum does not intersect sphere->discard object
      }else {
        //- sphere and object are (partially) in frustum: do not cull
      }
    }
  }

  l3d_object_clippable *oc;
```

```
    oc = dynamic_cast <l3d_object_clippable *>(objects[iObj]);
    if(oc) {
      oc->clip_to_plane(frustum.near);
      oc->clip_to_plane(frustum.far);
      oc->clip_to_plane(frustum.top);
      oc->clip_to_plane(frustum.bottom);
      oc->clip_to_plane(frustum.left);
      oc->clip_to_plane(frustum.right);
    }

    objects[iObj]->camera_transform(camera->viewing_xform);

    l3d_polygon_3d_node *n;

    //- vector from facet center to camera (at 0,0,0), for backface culling
    l3d_vector facet_to_cam_vec;
    //- facet's sfc normal relative to origin (aFacet.sfc_normal is relative
    //- to facet center)
    l3d_vector N;
    n = objects[iObj]->nonculled_polygon_nodes;
    while(n) {
      N = n->polygon->sfcnormal.transformed - n->polygon->center.transformed;

      facet_to_cam_vec.set (0 - n->polygon->center.transformed.X_,
                            0 - n->polygon->center.transformed.Y_,
                            0 - n->polygon->center.transformed.Z_,
                            int_to_l3d_real(0));

      //- dot product explicitly formulated so no fixed point overflow
      //- (facet_to_cam_vec is very large)
      //- (use of the inline function "dot" uses default fixed precision)

#include "../../math/fix_lowp.h"
#define BACKFACE_EPS float_to_l3d_real(0.1)
      if(( l3d_mulrr(FX_CVT(N.X_), FX_CVT(facet_to_cam_vec.X_)) +
           l3d_mulrr(FX_CVT(N.Y_), FX_CVT(facet_to_cam_vec.Y_)) +
           l3d_mulrr(FX_CVT(N.Z_), FX_CVT(facet_to_cam_vec.Z_)) > BACKFACE_EPS)
         //- equivalent to: if(N.dot(facet_to_cam_vec) > BACKFACE_EPS )
#include "../../math/fix_prec.h"

        )

      {
        //- polygon is not culled
      }else {

        //- take this polygon out of the list of nonculled_polygon_nodes
        if(n->prev) {
          n->prev->next = n->next;
        }else {
          objects[iObj]->nonculled_polygon_nodes = n->next;
        }

        if(n->next) {
          n->next->prev = n->prev;
        }
      }

      n = n->next;
    }
```

```
    objects[iObj]->apply_perspective_projection(*camera,
        screen->xsize, screen->ysize);

    n = objects[iObj]->nonculled_polygon_nodes;

    while(n) {
      {

        plist[pnum++] = n->polygon;

        /* Use the average of all z-values for depth sorting */
        l3d_real zsum=int_to_l3d_real(0);
        register int ivtx;

        for(ivtx=0; ivtx<n->polygon->ivertices->num_items; ivtx++) {
          zsum += (**(n->polygon->vlist))[ (*n->polygon->ivertices)[ivtx].ivertex
                    ].transformed_intermediates[1].Z_ ;
        }
        n->polygon->zvisible = l3d_divri(zsum, ivtx);
      }
      n = n->next;
    }
  }

  /* Sort the visible facets in decreasing order of z-depth */
  qsort(plist, pnum, sizeof(l3d_polygon_3d *), compare_facet_zdepth);

  /* Display the facets in the sorted z-depth order (painter's algorithm) */

  for(int i=0; i<pnum; i++) {
    plist[i]->draw(rasterizer);
  }

  char msg[256];
  sprintf(msg,"%d POLYS DRAWN", pnum);
  rasterizer->draw_text(0,16,msg);

  sprintf(msg,"%d objects culled",
          culled_obj_count);
  rasterizer->draw_text(0,32,msg);
}
```

The first thing we do in the new `draw_all` routine is compute the view frustum in world coordinates. We perform a reverse projection to get the four corner points on the near plane (using the formulas from Chapter 2), then call `l3d_viewing_frustum::create_from_points` to create the view frustum, passing the inverse camera transformation matrix as a parameter so that the frustum is computed in world space, not camera space. The `XOFF` and `YOFF` constants in the code are pixel offsets from the corner of the screen. Instead of using the extreme upper/lower left/right corners of the screen for the reverse projection, we use points slightly offset towards the center of the screen. For instance, instead of (0,0) at the upper left, we use (2,2). This makes the frustum slightly smaller than the actual screen, to allow for minor numerical inaccuracy in the clipping code. The reason we make the frustum smaller is that clipping polygons against the frustum is mathematically equivalent to 2D clipping to the view window, as we noted earlier. But smaller numerical errors in clipping computations can lead to clipped vertices which lie slightly outside the frustum. By artificially making the frustum smaller with the `XOFF` and `YOFF`

constants, we can be reasonably sure that even if a clipped vertex in 3D lies slightly outside the frustum, it will, when projected, still lie within the 2D screen boundaries—which is important if we use 3D frustum clipping as a replacement for 2D clipping.

Then, we perform view frustum culling on an object with the following snippet of code:

```
l3d_object_clippable_boundable *obj;
if((obj=dynamic_cast<l3d_object_clippable_boundable *> (objects[iObj])))
{

  //- first, position *ONLY* the bounding sphere in world space
  obj->bounding_sphere.transform(objects[iObj]->modeling_xform);

  //- now, check if the bounding sphere is in the frustum
  if(! frustum.intersects_sphere(obj->bounding_sphere)) {
    culled_obj_count++;
    continue; //- frustum does not intersect sphere->discard object
  }else {
    //- sphere is at least partially in frustum. so, now,
    //- transform *ALL* of the object's vertices into world space.

    objects[iObj]->transform(objects[iObj]->modeling_xform);
  }
}
```

First, we transform just the bounding sphere of the object from object space into world space. Then, we check to see if the bounding sphere is outside of the frustum; if it is, we discard the object and continue processing the next object in the world. Otherwise, if the bounding sphere was at least partially inside the frustum, we transform the geometry bounded by the sphere into world space, and continue processing the object as normal. The actual code has two main `if` branches, one for the case that the object has a transformation matrix from object space to world space, and one for the case that the object has no transformation (i.e., the object's coordinates are already world coordinates), but the general logic—if bounding sphere in frustum, then keep, else reject—is the same for both cases.

If the object's bounding sphere is in the view frustum, then the next step is to clip the object in 3D against the view frustum planes:

```
l3d_object_clippable *oc;
oc = dynamic_cast <l3d_object_clippable *>(objects[iObj]);
if(oc) {
  oc->clip_to_plane(frustum.near);
  oc->clip_to_plane(frustum.far);
  oc->clip_to_plane(frustum.top);
  oc->clip_to_plane(frustum.bottom);
  oc->clip_to_plane(frustum.left);
  oc->clip_to_plane(frustum.right);
}
```

The rest of the object processing is identical to what we have seen before: transformation into camera space, back-face culling, perspective projection, polygon sorting, and rasterization. (The next section, which is on the painter's algorithm, covers the topic of polygon sorting in more detail.) Notice that we do not clip polygons in 2D to the view window in this case; 3D view frustum clipping means that any projected polygons will lie within the view window (except in the case of numerical inaccuracy, as mentioned earlier). Finally, the new `draw_all` routine also draws a text string to the display, indicating the number of objects culled.

The view frustum culling as implemented here, however, still has problems as the number of objects increases. We perform culling at the object level, with no further hierarchy. This means that as the number of objects increases, we must linearly search through more and more objects, testing each one to see if it is in the frustum or not. Even if we can save work by not processing the objects further, it still costs more and more time to check more and more objects. As we saw earlier, by hierarchically grouping objects into larger and larger groups, and by first checking these large groups against the view frustum, we can save time by discarding large groups from further processing without needing to traverse every node in the hierarchy.

The problem remains of how to generate such a hierarchy. You could hard-code the hierarchy into your program, but this would be very difficult to maintain. You could use the 3D features of the 3D modeling package to impose hierarchy on your scene, assuming you could export this information to be read in by a 3D program. This still has the problem that any changes to the scene would require changing or updating the hierarchy within the 3D modeler. A very interesting option is the use of a BSP tree to automatically generate a hierarchy of bounding volumes for an arbitrary set of polygons, as discussed in the next chapter.

But before talking about BSP trees, let's continue looking at some simpler VSD algorithms. Next, we cover a simple and popular algorithm, which we already have seen and used: the painter's algorithm.

The Painter's Algorithm

The painter's algorithm is a VSD algorithm, relying on polygon sorting, which addresses the correctness issue but neglects the efficiency issue. We know that drawing polygons in random order to the screen sometimes results in an incorrect display because farther polygons obscure nearer ones. The solution adopted in the original `l3d_world` class was the so-called painter's algorithm, where we draw polygons from back-to-front order into the frame buffer. By drawing the nearest polygons last, the pixels for the nearest polygons overwrite any pixels from farther polygons which might already be in the frame buffer at the same positions. This means that nearer polygons always obscure farther polygons.

Since the sample programs using the `l3d_world` class already use the painter's algorithm, there is no additional sample program here. The painter's algorithm is very simple, and produces mostly correct results most of the time. However, there are a few important shortcomings and observations about the painter's algorithm which we should take note of.

- Sorting. The painter's algorithm requires us to sort polygons based on their distance to the viewer. Sorting takes time. For large numbers of polygons, the painter's algorithm can spend much of its time sorting the polygons. One technique to avoid such sorting is to create a large list of polygons, say large enough for 4096 polygons. Each entry in this list represents one z depth. Then, before drawing a polygon, we scale its z depth to lie within the range of the list. We round this scaled z depth to an integer and use it as the polygon's insertion position into the list. If two polygons' z depths are close enough, they get assigned to the same slot, which can result in small-scale sorting errors. Then, to draw the polygons, we traverse the list from back

to front. In essence, this scheme is a radix sort, using the *z* depth itself as an index into the list to avoid sorting.

- Overdraw. The painter's algorithm suffers from *overdraw*. Overdraw means that a particular pixel is often drawn more than once into the frame buffer. Consider a set of 500 texture mapped polygons, all facing the viewer and positioned one behind the other. With the painter's algorithm, we first draw the farthest polygon, taking the time to transform it into camera space, do perspective division and clipping, and carry out the entire texture mapping computations for every single pixel of the polygon. Then, we draw the second farthest polygon, again performing many computations for every pixel, entirely overwriting the results of the farthest polygon. Then, we draw the third farthest polygon, again entirely overwriting the results of the first two polygons. Although this example is extreme, it shows how a back-to-front drawing of polygons wastes time by carefully computing the correct pixels and colors to be drawn, then possibly overwriting much or all of this work when a nearer polygon is drawn. The painter's algorithm works because of overdraw, but when the overdrawn pixels take a significant amount of time to compute—as is the case with texture mapping—more overdraw means more "wasted" time computing pixels which will be overwritten anyway. It is important to note, however, that the overdraw concern is mainly an issue for software rendering. With hardware-accelerated rendering, it is often possible to completely ignore this issue of overdraw, since the hardware can draw polygons so quickly.
- Non-orderable polygon groups. The painter's algorithm does not produce correct results all of the time. One classic case is cyclic overlap. If a series of polygons overlap such that they form a cycle (see Figure 4-15), then there is no single ordering which can be applied to draw the polygons so that a correct display results. Similarly, if one polygon pierces another polygon, then there is also no single ordering which is correct: part of the piercing polygon lies in front of, part behind the pierced polygon. In both of these cases, the painter's algorithm can still be used if we perform a number of tests and polygon splitting operations. Splitting polygons into more pieces resolves any ambiguities in the *z* ordering, producing more independent polygons to be drawn, but which can then be drawn in a correct order. The tests to be done for the splitting are a bit tedious, and in my opinion, not really worth the trouble; we won't cover them further here. Part of the appeal of the painter's algorithm is its simplicity, and if we start to bog down the code with several special case tests, then it might be better just to use a different VSD algorithm entirely.

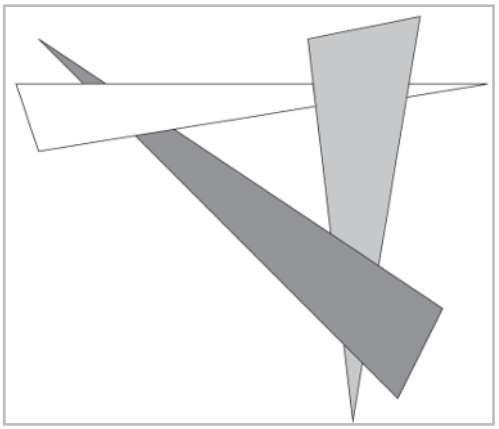


Figure 4-15: Cyclic overlap cannot be resolved by the simple painter's algorithm.

TEAM-FLY

You can see a good example of the kinds of problems which arise with the painter's algorithm in the program `textest` from Chapter 2. Here, we had two pyramid objects, one static and one rotating. The rotating pyramid object pierced the static pyramid object. If you look closely at the appearance of the piercing pyramid, you will notice that the polygons instantaneously "flip" from in front of the pierced pyramid to behind it. This is because the simple painter's algorithm tries to find a single sort order to draw all polygons, based on the polygon's average z value. As soon as the average z value is smaller, the polygon appears in front; as soon as the average z value is greater, the polygon appears behind the others. In the case of a piercing polygon, as we said before, this is not correct: part of the piercing polygon lies in front of, part behind the pierced polygon. See Figures 4-16 and 4-17: the piercing polygon, highlighted with a white border, is always drawn either completely in front of or completely behind the pierced polygons. When observing the animation in the running program, the polygon appears to flip from one side to the other.

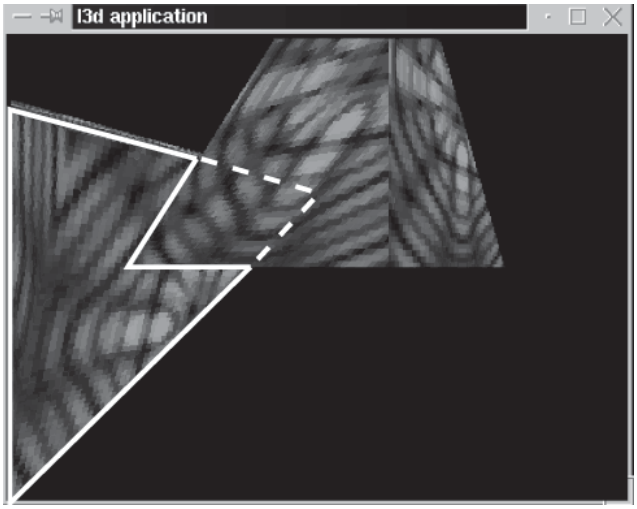


Figure 4-16: The highlighted polygon is drawn completely behind the static pyramid.

Team-Fly®

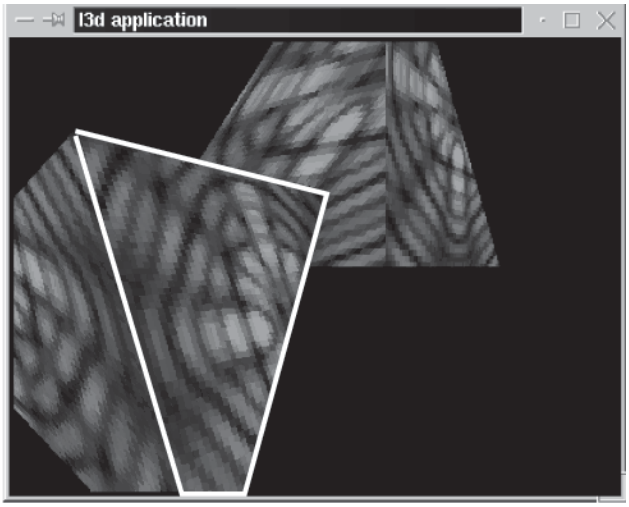


Figure 4-17: The highlighted polygon is drawn completely in front of the static pyramid.

Now, let's look at a simple VSD routine which can solve these sorting problems of the painter's algorithm: the *z* buffer algorithm.

The Z Buffer Algorithm

The *z* buffer algorithm is a per-pixel VSD algorithm which mainly addresses the VSD correctness issue, essentially ignoring the VSD efficiency issue. The idea is very simple. We maintain a 2D array of *z* values, which has the same dimensions as the rasterizer buffer. This array is called a *z buffer*. Each position in the *z* buffer corresponds to a pixel in the rasterizer buffer. Furthermore, each value in the *z* buffer represents the *z* value (or a $1/z$ value, as we see shortly) of the point in 3D space whose projection yields the pixel at that location in the *z* buffer.

We use the *z* buffer as follows. At the beginning of each frame, before any objects are rasterized, we clear the *z* buffer by setting all of its values to represent the maximum possible distance. Then, we rasterize polygons as normal, pixel for pixel. When rasterizing polygons, for each pixel we compute the *z* value of the current pixel being rasterized. Then, we compare the *z* value for the current pixel against the *z* value already in the *z* buffer at the pixel's location.

If the current *z* value has a smaller depth than the old *z* buffer value, then the 3D point corresponding to the pixel being drawn lies closer to the viewer than the 3D point corresponding to the pixel that was previously drawn. Thus, the new pixel must obscure the old pixel. We draw the new pixel and update the *z* buffer at that pixel position with the new depth.

If, on the other hand, the current *z* value has a greater depth than the old *z* buffer value, then this means that the 3D point corresponding to the pixel being drawn lies farther from the viewer than the 3D point corresponding to the pixel that was previously drawn. Thus, the old pixel obscures the new pixel. We do not draw the pixel, we do not update the *z* buffer, and we continue rasterization of the next pixel in the polygon.

The *z* buffer algorithm is accurate to the pixel. It correctly handles polygons that pierce one another, as well as cyclic overlap. It can even be used for non-polygonal objects; all that is needed is some way of determining the *z* value of the object for a given pixel position.

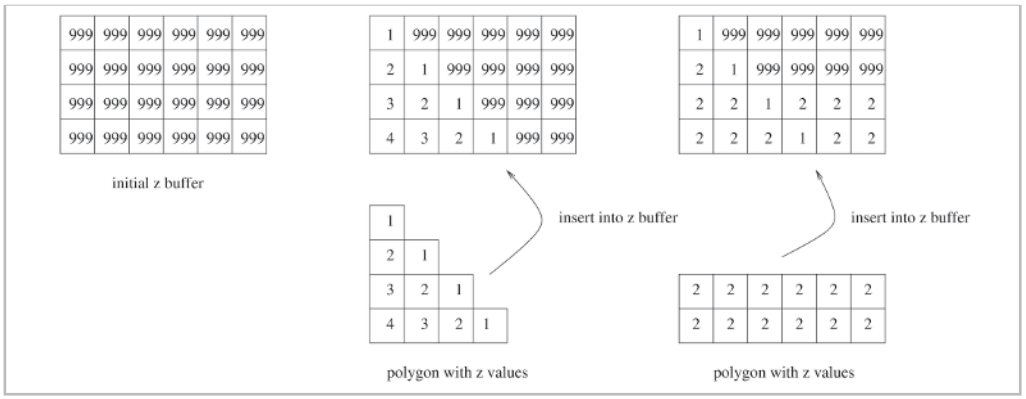


Figure 4-18: The z buffer. At first, it is initialized to the maximum depth value (left). Then we insert one polygon, drawing each pixel and storing its depth values into the z buffer only if the polygon's z value is less than that in the z buffer (middle). Finally we insert another polygon, whose z values again are compared with the existing z values in the z buffer (right).

General Observations about the Z Buffer

Let's now look at some important aspects of the *z* buffer algorithm.

- Sorting. We do not need to sort objects before sending them to the *z* buffer. Eventually, after all objects are rasterized, the nearest pixels will correctly obscure the farthest pixels. However, if we do perform a rough front-to-back sort, this can save time. This is because drawing polygons in back-to-front order into the *z* buffer means that many painstakingly computed pixels will be wastefully overwritten, just as with the painter's algorithm. But by drawing polygons roughly front to back, the nearest *z* values will get written first into the *z* buffer, meaning that pixels for the later, farther polygons will lie behind the existing *z* buffer values and thus do not need to be computed. The front-to-back sorting can be "rough" and does not need to be exact, because the *z* buffer ultimately compares depth at the pixel level; the time needed to perform a full-blown painter's style sort on all polygons is not needed. A rough front-to-back sorting, for instance, might be a sorting of objects from front to back, but not of the polygons within the objects.
- Memory. The *z* buffer needs to be as large as the rasterizer's frame buffer; thus, we are keeping two copies of the screen information. Each *z* value in the *z* buffer needs to be stored with enough bytes of accuracy to ensure an accurate representation of the *z* values. For instance, storing each *z* value in the *z* buffer with only one byte would mean that we could only have 256 different depth values, which is far too few for an accurate *z* buffer. Typically we use between 2 bytes (16 bits) and 4 bytes (32 bits) to store each *z* value within the *z* buffer.
- Speed. Before each frame, the *z* buffer needs to be reset by setting each value to the maximum possible depth. Since the *z* buffer is just as large as the rasterizer buffer, this means that we

have to perform an initialization of another large memory buffer (the frame buffer also must be cleared), at the beginning of each frame, which costs time. Furthermore, the z value comparisons take place per pixel within the innermost loop of the rasterizer, where we must determine for each pixel if it is to be drawn or not. Adding this comparison to the inner loop of the rasterizer can cause a great performance degradation in a software rasterizer.

- Artifacts. Regardless of how many bits of precision we use to store the z values in the z buffer, there are always situations where the z buffer accuracy is not enough. Consider a 3D model where a sheet of paper lies on top of a desk. In reality, the sheet of paper lies perhaps 0.000005 meters above the surface of the desk. Due to numerical inaccuracy inherent in finite-precision computations, it is possible that when rasterizing the sheet of paper, the computed z values incorrectly lie very slightly behind those of the underlying desk. This would cause the desk model to "poke through" the sheet of paper wherever this occurs, which is rather disturbing. The only solution to such accuracy problems is to alter the position of the models slightly so that the distance between them is great enough to be differentiated by the limited precision of the z buffer.
- z value computations. We've mentioned so far that we store in the z buffer the z value of the 3D point whose projection yields the 2D location of the pixel being rasterized. We can compute this value by taking a reverse projection, using the pixel location and the plane equation of the polygon. This gives us the exact z value for the pixel in question. However, the reverse projection operation requires one divide for every pixel. Fundamentally, this is because z values are not linear in 2D screen space due to the perspective projection, as we saw in Chapter 2. But, as we also observed in Chapter 2, $1/z$ values are linear in screen space. With a z buffer, the actual z value is not important; it is the relationship among the various z values that is important. With this understanding, we can use a $1/z$ buffer instead of a z buffer and achieve the same result, without needing to perform the per-pixel division of a reverse projection to retrieve the 3D z value. Instead, with a $1/z$ buffer, we take the $1/z$ values at each polygon's vertex and linearly interpolate them in 2D across the polygon. Conveniently, we already need to perform this operation anyway, for texture mapping! Then, we store the $1/z$ values in the z buffer, without performing a divide to retrieve the original z. We also reverse the sense of the comparison operation; instead of checking if the current pixel's z value is less than the z buffer value, we check to see if the current pixel's $1/z$ value is greater than the z buffer value. Finally, we clear the z buffer to 0 instead of the maximum distance, because with a $1/z$ buffer, $1/0$ (infinity) is effectively the maximum distance. In this way, we preserve the relative ordering of the z values, though we are actually storing and comparing $1/z$ values to avoid the per-pixel division needed to retrieve the original z.
- Fog. As mentioned in Chapter 2, fog can be computed as a function of the z distance of a pixel from the viewer, although the radial distance would be the correct distance to use. We now see why the z distance is often used instead: with a z buffer, we already have the z values, for every pixel, which can be used to create a simple, albeit somewhat inaccurate, fog effect.

A Software Z Buffer: Class l3d_rasterizer_3d_zbuf_sw_imp

Class `l3d_rasterizer_3d_zbuf_sw_imp` is the l3d software rasterizer implementing *z* buffering. It draws textured polygons into a *z* buffer. The implementation is a fairly simple extension of the textured polygon drawing routine. Since we already need to interpolate the 1/*z* values across the polygon for texture mapping, all we need to do for *z* buffering is to store these values in a *z* buffer, and to compare each pixel's 1/*z* value against the 1/*z* value in the *z* buffer to see if the current pixel should be drawn or not.

Listing 4-13: `ras3z_sw.h`

```
#ifndef __RAS3Z_SW_H
#define __RAS3Z_SW_H
#include "../tool_os/memman.h"

#include "ras3_sw.h"
#include "math.h"
#include "../system/sys_dep.h"

class l3d_rasterizer_3d_zbuf_sw_imp :
      virtual public l3d_rasterizer_3d_sw_imp
{
  private:
    l3d_fixed *zbuf;
    long int zbuf_size;
  public:
    l3d_rasterizer_3d_zbuf_sw_imp(int xs, int ys, l3d_screen_info *si);
    virtual ~l3d_rasterizer_3d_zbuf_sw_imp(void);

    /* virtual */ void draw_polygon_textured(const l3d_polygon_3d_textured *p_poly);

    /* virtual */
    void clear_buffer(void);
};

class l3d_rasterizer_3d_zbuf_sw_imp_factory :
      public l3d_rasterizer_3d_imp_factory
{
  public:
    l3d_rasterizer_3d_imp *create(int xs, int ys, l3d_screen_info *si);
};

#endif
```

Listing 4-14: `ras3z_sw.cc`

```
#include "ras3z_sw.h"
#include "../system/sys_dep.h"
#include <string.h>
#include <stdlib.h>
#include <values.h>
#include "../tool_os/memman.h"

l3d_rasterizer_3d_imp * l3d_rasterizer_3d_zbuf_sw_imp_factory::create
(int xs, int ys, l3d_screen_info *si)
{
  return new l3d_rasterizer_3d_zbuf_sw_imp(xs,ys,si);
}
```

```
 l3d_rasterizer_3d_zbuf_sw_imp::l3d_rasterizer_3d_zbuf_sw_imp
(int xs, int ys, l3d_screen_info *si)
    :
    l3d_rasterizer_3d_sw_imp(xs,ys,si),
    l3d_rasterizer_3d_imp(xs,ys,si),
    l3d_rasterizer_2d_sw_imp(xs,ys,si),
    l3d_rasterizer_2d_imp(xs,ys,si)
{
  zbuf = new l3d_fixed[xs*ys];
  zbuf_size = xs*ys;
  long register int i;
  for(i=0; i<zbuf_size; i++) {
    zbuf[i] = 0;
  }
}

l3d_rasterizer_3d_zbuf_sw_imp::~l3d_rasterizer_3d_zbuf_sw_imp(void) {
  delete [] zbuf;
}

void l3d_rasterizer_3d_zbuf_sw_imp::clear_buffer(void) {
  l3d_rasterizer_3d_sw_imp::clear_buffer();

  long register int i;

  for(i=0; i<zbuf_size; i++) {
    zbuf[i] = 0;
  }
}

#define Z_MULT_FACTOR 512
#define Z_ADD_FACTOR -0.42

//- convenience macro for accessing the vertex coordinates. Notice
//- that we access the clipped vertex index list (not the original),
//- and the transformed coordinate (not the original). This means
//- we draw the clipped version of the transformed polygon.
#define VTX(i) ((**(p_poly->vlist))[ (*(p_poly->clip_ivertices))[i].ivertex ].transformed)

#include "../math/fix_lowp.h"
#if 0
#define l3d_fixed float
#define l3d_real_to_fixed(x) (x)
#define fixfixdiv(a,b) ((a)/(b))
#define fixfixmul(a,b) ((a)*(b))
#define int2fix(a) ( (float)(a))
#define fix2int(a) ( (int)(a))
#define fix2float(a) (a)
#define float2fix(a) (a)
#define iceil_fix(a) ( (int)ceil((double)(a)) )

#endif

void l3d_rasterizer_3d_zbuf_sw_imp::draw_polygon_textured
(const l3d_polygon_3d_textured *p_poly)
{

  l3d_fixed x0,y0,x1,y1,x2,y2,x3;
  l3d_fixed top_y, bottom_y;
  int point_on_right=0;
```

```
int left_idx, right_idx, top_y_idx, bottom_y_idx;

int maxy_upper, iceil_fix_y0, iceil_fix_y1, iceil_fix_y2;

l3d_point *vtemp;

int i;

int scanline;

//- variables for the left edge, incremented BETWEEN scanlines
//- (inter-scanline)

l3d_fixed volatile
left_x,                    left_ui,       left_vi,       left_zi,
left_x_start, left_y_start, left_ui_start, left_vi_start, left_zi_start,
left_x_end,   left_y_end,   left_ui_end,   left_vi_end,   left_zi_end,
left_dx_dy,                left_dui_dy,   left_dvi_dy,   left_dzi_dy;
int
left_ceilx,
left_ceilx_start, left_ceilx_end,
left_ceily_start, left_ceily_end;

//- variables for the right edge, incremented BETWEEN scanlines
//- (inter-scanline)

l3d_fixed volatile
right_x,                    right_ui,      right_vi,      right_zi,
right_x_start,right_y_start,right_ui_start,right_vi_start,right_zi_start,
right_x_end,  right_y_end,  right_ui_end,  right_vi_end,  right_zi_end,
right_dx_dy,                right_dui_dy,  right_dvi_dy,  right_dzi_dy;

int
right_ceilx,
right_ceilx_start, right_ceilx_end,
right_ceily_start, right_ceily_end;

//- variables incremented WITHIN one scanline (intra-scanline)

l3d_fixed volatile
u,v,z,
ui, vi, zi, dui_dx, dvi_dx, dzi_dx,

u_left, u_right, du_dx,
v_left, v_right, dv_dx,
inv_dx, denom, inv_run_dx,

run_x, run_x_start, run_x_end,
u_run_end,  v_run_end, du_dx_run, dv_dx_run;

long int cur_x;

top_y = l3d_real_to_fixed(VTX(0).Y_);
top_y_idx = 0;
bottom_y = top_y;
bottom_y_idx = top_y_idx;
for(i=0; i<p_poly->clip_ivertices->num_items; i++) {
  if(l3d_real_to_fixed(VTX(i).Y_) < top_y) {
    top_y = l3d_real_to_fixed(VTX(i).Y_);
    top_y_idx = i;
```

```
    }
    if(l3d_real_to_fixed(VTX(i).Y_) > bottom_y) {
      bottom_y = l3d_real_to_fixed(VTX(i).Y_);
      bottom_y_idx = i;
    }
  }

  left_idx = top_y_idx;
  right_idx = top_y_idx;

  left_x_start=l3d_real_to_fixed(VTX(top_y_idx).X_);
  left_ceilx_start=iceil_fix(left_x_start);
  left_y_start=l3d_real_to_fixed(VTX(top_y_idx).Y_);
  left_ceily_start=iceil_fix(left_y_start);
  left_ceily_end=left_ceily_start;
  left_zi_start = fixfixdiv(int2fix(Z_MULT_FACTOR),
                    l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                       & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Z_)
                          + float2fix(Z_ADD_FACTOR));
  left_zi_end = left_zi_start;
  left_ui_start =
    fixfixmul(
      l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                         & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.X_),
      left_zi_start);
  left_ui_end = left_ui_start;
  left_vi_start = fixfixmul(
                    l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                       & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Y_),
                    left_zi_start);
  left_vi_end = left_vi_start;

  right_x_start=left_x_start;
  right_y_start=left_y_start;
  right_ceily_start=left_ceily_start;
  right_ceily_end=right_ceily_start;
  right_ui_start = left_ui_start;
  right_vi_start = left_vi_start;
  right_zi_start = left_zi_start;

  scanline = left_ceily_start;
  int oneshot=1;

  while(scanline < ysize) {
    while( left_ceily_end - scanline <= 0 )
    {
      if (left_idx == bottom_y_idx) {
        return;
      }
      left_idx = p_poly->next_clipidx_left(left_idx,p_poly->clip_ivertices->num_items);
      left_y_end=l3d_real_to_fixed(VTX(left_idx).Y_);
      left_ceily_end = iceil_fix(left_y_end);
      if(left_ceily_end - scanline) {
#define MIN_EDGE_HEIGHT float2fix(0.005)
#define MIN_EDGE_WIDTH float2fix(0.005)
        left_x_end=l3d_real_to_fixed(VTX(left_idx).X_);
        left_ceilx_end=iceil_fix(left_x_end);

        left_zi_end = fixfixdiv(int2fix(Z_MULT_FACTOR),
                        l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
```

```
                                        & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Z_)
                              + float2fix(Z_ADD_FACTOR));
    left_ui_end = fixfixmul(
                  l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                        & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.X_),
                  left_zi_end);
    left_vi_end = fixfixmul(
                  l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                        & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Y_),
                  left_zi_end);

    if(left_y_end - left_y_start >= MIN_EDGE_HEIGHT) {
      left_dx_dy = fixfixdiv(left_x_end-left_x_start,left_y_end-left_y_start);
      left_dui_dy = fixfixdiv(left_ui_end - left_ui_start,
                              left_y_end  - left_y_start);
      left_dvi_dy = fixfixdiv(left_vi_end - left_vi_start,
                              left_y_end  - left_y_start);
      left_dzi_dy = fixfixdiv(left_zi_end - left_zi_start,
                              left_y_end  - left_y_start);
    }
    else
    {
      left_dx_dy =
        left_dui_dy =
          left_dvi_dy =
            left_dzi_dy = int2fix(0);
    }

    left_x = left_x_start +
             fixfixmul(int2fix(left_ceily_start)-left_y_start , left_dx_dy);
    left_ui = left_ui_start + //- sub-pixel correction
              fixfixmul(int2fix(left_ceily_start)-left_y_start,
                        left_dui_dy);
    left_vi = left_vi_start + //- sub-pixel correction
              fixfixmul(int2fix(left_ceily_start)-left_y_start,
                        left_dvi_dy);
    left_zi = left_zi_start + //- sub-pixel correction
              fixfixmul(int2fix(left_ceily_start)-left_y_start,
                        left_dzi_dy);
  }else {
    left_x_start = l3d_real_to_fixed(VTX(left_idx).X_);
    left_ceilx_start=iceil_fix(left_x_start);
    left_y_start = l3d_real_to_fixed(VTX(left_idx).Y_);
    left_ceily_start = iceil_fix(left_y_start);
    left_zi_start = fixfixdiv(int2fix(Z_MULT_FACTOR),
                      l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                          & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Z_)
                            + float2fix(Z_ADD_FACTOR));
    left_ui_start = fixfixmul(
                    l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                        & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.X_),
                    left_zi_start);
    left_vi_start = fixfixmul(
                    l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                        & ((*(p_poly->clip_ivertices))[left_idx]))->tex_coord.Y_),
                    left_zi_start);

  }
}
```

```
while(right_ceily_end - scanline <= 0 )
{
  if (right_idx == bottom_y_idx) {
    return;
  }
  right_idx = p_poly->next_clipidx_right(right_idx, p_poly->clip_ivertices->num_items);
  right_y_end=l3d_real_to_fixed(VTX(right_idx).Y_);
  right_ceily_end = iceil_fix(right_y_end);
  if(right_ceily_end - scanline) {

    right_x_end=l3d_real_to_fixed(VTX(right_idx).X_);
    right_ceilx_end = iceil_fix(right_x_end);
    right_zi_end = fixfixdiv(int2fix(Z_MULT_FACTOR),
                    l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                  & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.Z_)
                           + float2fix(Z_ADD_FACTOR));
    right_ui_end = fixfixmul(
                    l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                  & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.X_),
                    right_zi_end);
    right_vi_end = fixfixmul(
                    l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                  & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.Y_),
                    right_zi_end);

    if(right_y_end-right_y_start>=MIN_EDGE_HEIGHT) {
      right_dx_dy =
        fixfixdiv(right_x_end-right_x_start,right_y_end-right_y_start);
      right_dui_dy = fixfixdiv(right_ui_end - right_ui_start,
                               right_y_end  - right_y_start);
      right_dvi_dy = fixfixdiv(right_vi_end - right_vi_start,
                               right_y_end  - right_y_start);
      right_dzi_dy = fixfixdiv(right_zi_end - right_zi_start,
                               right_y_end  - right_y_start);
    }
    else
    {
      right_dx_dy =
        right_dui_dy =
          right_dvi_dy =
            right_dzi_dy = int2fix(0);
    }

    right_x = right_x_start +
              fixfixmul(int2fix(right_ceily_start)-right_y_start , right_dx_dy);

    right_ui = right_ui_start + //- sub-pixel correction
               fixfixmul(int2fix(right_ceily_start)-right_y_start,
                         right_dui_dy);
    right_vi = right_vi_start + //- sub-pixel correction
               fixfixmul(int2fix(right_ceily_start)-right_y_start,
                         right_dvi_dy);
    right_zi = right_zi_start + //- sub-pixel correction
               fixfixmul(int2fix(right_ceily_start)-right_y_start,
                         right_dzi_dy);

  }else {
    right_x_start = l3d_real_to_fixed(VTX(right_idx).X_);
    right_ceilx_start = iceil_fix(right_x_start);
    right_y_start = l3d_real_to_fixed(VTX(right_idx).Y_);
```

```
        right_ceily_start = iceil_fix(right_y_start);

        right_zi_start = fixfixdiv(int2fix(Z_MULT_FACTOR),
                          l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                  & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.Z_)
                                 +float2fix(Z_ADD_FACTOR));
        right_ui_start = fixfixmul(
                          l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                  & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.X_),
                          right_zi_start);
        right_vi_start = fixfixmul(
                          l3d_real_to_fixed(((l3d_polygon_ivertex_textured *)
                                  & ((*(p_poly->clip_ivertices))[right_idx]))->tex_coord.Y_),
                          right_zi_start);
      }
    }

    if (left_ceily_end > ysize) left_ceily_end = ysize;
    if (right_ceily_end > ysize) right_ceily_end = ysize;

    while ( (scanline < left_ceily_end) && (scanline < right_ceily_end) )
    {
      if (left_x > right_x) {
      }
      else {
        left_ceilx = iceil_fix(left_x);
        right_ceilx = iceil_fix(right_x);

        if(right_x - left_x > MIN_EDGE_WIDTH) {
          dui_dx = fixfixdiv(right_ui - left_ui , right_x  - left_x);
          dvi_dx = fixfixdiv(right_vi - left_vi , right_x  - left_x);
          dzi_dx = fixfixdiv(right_zi - left_zi , right_x  - left_x);
        }else {
          dui_dx =
            dvi_dx =
              dzi_dx = int2fix(0);
        }

        ui = left_ui + //- sub-pixel correction
             fixfixmul(int2fix(left_ceilx) - left_x, dui_dx);
        vi = left_vi + //- sub-pixel correction
             fixfixmul(int2fix(left_ceilx) - left_x, dvi_dx);
        zi = left_zi + //- sub-pixel correction
             fixfixmul(int2fix(left_ceilx) - left_x, dzi_dx);

        cur_x=left_ceilx;

#define LINEAR_RUN_SHIFT 4
#define LINEAR_RUN_SIZE (1<<LINEAR_RUN_SHIFT)
        z = fixfixdiv(int2fix(1), zi);
        u = fixfixmul(ui,z);
        v = fixfixmul(vi,z);

        l3d_fixed *zbuf_ptr;

        unsigned char *pix;
        for(pix=sinfo->p_screenbuf+(left_ceilx + SW_RAST_Y_REVERSAL(ysize,
            scanline)*xsize)*sinfo->bytes_per_pixel,
            zbuf_ptr = zbuf + (left_ceilx + scanline*xsize)
                       ;
```

TEAMFLY

```
    pix< sinfo->p_screenbuf+(right_ceilx+ SW_RAST_Y_REVERSAL(ysize,
                              scanline)*xsize)*sinfo->bytes_per_pixel;
   )
{
  run_x_end = int2fix(cur_x) +
              int2fix(LINEAR_RUN_SIZE);
  if (run_x_end > int2fix(right_ceilx)) {
    run_x_end = int2fix(right_ceilx);
  }
  denom = fixfixdiv
          (int2fix(1) ,
           zi + fixfixmul(dzi_dx, run_x_end-int2fix(cur_x)));
  u_run_end=fixfixmul
            (ui + fixfixmul(dui_dx,(run_x_end-int2fix(cur_x))),
             denom) ;
  v_run_end=fixfixmul
            (vi + fixfixmul(dvi_dx,(run_x_end-int2fix(cur_x))),
             denom) ;

  inv_run_dx = fixfixdiv
               (int2fix(1),
                (run_x_end-int2fix(cur_x)));
  du_dx_run = fixfixmul((u_run_end - u) ,inv_run_dx);
  dv_dx_run = fixfixmul((v_run_end - v) ,inv_run_dx);

  for(run_x = int2fix(cur_x);
      run_x < run_x_end;
      run_x+=int2fix(1))
  {
    unsigned char *texel =
      p_poly->texture->tex_data->data +
      ((fix2int(l3d_mulri(v,p_poly->texture->tex_data->height-1))&
      (p_poly->texture->tex_data->height-1))*(p_poly->texture->tex_data->width)
       +
      (fix2int(l3d_mulri(u,p_poly->texture->tex_data->width-1))&
      (p_poly->texture->tex_data->width-1)) ) *
      sinfo->bytes_per_pixel;

    if(zi > *zbuf_ptr) {
      *zbuf_ptr = zi;

      for(register int b=0; b<sinfo->bytes_per_pixel;b++) {
        *pix++ =  *texel++;
      }

      int fog = int2fix(Z_MULT_FACTOR) - fixintmul(zi,64);
      fog = fog >> 14;
      if(fog<0) fog=0;
      if(fog>MAX_LIGHT_LEVELS) fog=MAX_LIGHT_LEVELS;
      sinfo->fog_native(pix - sinfo->bytes_per_pixel, fog);
    }else {
      pix+= sinfo->bytes_per_pixel;
      texel+= sinfo->bytes_per_pixel;
    }
    zbuf_ptr++;
    u += du_dx_run;
    v += dv_dx_run;
    zi += dzi_dx;
  }
```

```
          cur_x += LINEAR_RUN_SIZE;
#if 0
          ui += dui_dx<<LINEAR_RUN_SHIFT;
          vi += dvi_dx<<LINEAR_RUN_SHIFT;
          zi += dzi_dx<<LINEAR_RUN_SHIFT;
#else
ui += l3d_mulri(dui_dx, LINEAR_RUN_SIZE);
          vi += l3d_mulri(dvi_dx, LINEAR_RUN_SIZE);
#endif
        }
      }

      scanline++;
      left_x += left_dx_dy;
      right_x += right_dx_dy;

      left_ui += left_dui_dy;
      left_vi += left_dvi_dy;
      left_zi += left_dzi_dy;
      right_ui += right_dui_dy;
      right_vi += right_dvi_dy;
      right_zi += right_dzi_dy;
    }

    if ( left_ceily_end - scanline <= 0 ) {
      left_x_start=left_x_end;
      left_y_start=left_y_end;
      left_ceily_start=left_ceily_end;
      left_ui_start=left_ui_end;
      left_vi_start=left_vi_end;
      left_zi_start=left_zi_end;

    }
    if ( right_ceily_end - scanline <= 0 ) {
      right_x_start=right_x_end;
      right_y_start=right_y_end;
      right_ceily_start=right_ceily_end;
      right_ui_start=right_ui_end;
      right_vi_start=right_vi_end;
      right_zi_start=right_zi_end;

    }
  }

}
#include "../math/fix_prec.h"
```

The constructor declares the *z* buffer as an array of fixed-point values, with the same dimensions as the frame buffer, and initializes all values in the buffer to be zero. The destructor deletes this array. The overridden `clear_buffer` method has been extended to clear not only the frame buffer, but also the *z* buffer.

The routine of interest is the overridden `draw_polygon_textured` routine. The routine is largely identical to the non-*z*-buffered version. As mentioned earlier, for texture mapping we are already computing a $1/z$ value for each pixel. All we need to do for a *z* buffer is to store these values into the *z* buffer and to compare against the existing values before drawing a pixel. The polygon is rasterized in horizontal spans. For each horizontal span of the polygon, we first

initialize the pointer `zbuf_ptr` to point to the entry in the *z* buffer corresponding to the first pixel in the span. Then, for each pixel we draw in the span, we check it against the *z* buffer, and update the *z* buffer if necessary. The following snippet is the *z* buffer check.

```
if(zi > *zbuf_ptr) {
  *zbuf_ptr = zi;

  for(register int b=0; b<sinfo->bytes_per_pixel;b++) {
    *pix++ =  *texel++;
  }

  int fog = int2fix(Z_MULT_FACTOR) - fixintmul(zi,64);
  fog = fog >> 14;
  if(fog<0) fog=0;
  if(fog>MAX_LIGHT_LEVELS) fog=MAX_LIGHT_LEVELS;
  sinfo->fog_native(pix - sinfo->bytes_per_pixel, fog);
}else {
  pix+= sinfo->bytes_per_pixel;
  texel+= sinfo->bytes_per_pixel;
}
zbuf_ptr++;
```



NOTE Actually the 1/z values computed within the software rasterizer are Z_MULT_FACTOR/(z+Z_ADD_FACTOR), to compensate for fixed-point accuracy problems, as we saw in Chapter 2. Still, for simplicity, we refer to these values as "1/z" values; the actual values have simply been offset and scaled slightly from the real 1/z value.

In the above loop, notice that after determining that a pixel is indeed visible, we also apply a fog computation to the pixel. Since we already know the 1/*z* value of the current pixel, calculating a *z*-based fog value is as simple as using the 1/*z* value in some formula which maps the 1/*z* value to a fog value, which we then use with the fog table of Chapter 2 to fade the color of the pixel into fog, similar to the way we used the lighting tables. Here, we simply multiply the 1/*z* value (which has actually been multiplied by Z_MULT_FACTOR) by 64 and subtract it from the maximum scaled 1/*z* value, which is Z_MULT_FACTOR. This has the effect of giving an integer fog value somewhere in the vicinity of the value of Z_MULT_FACTOR. Then we shift the fog value right by 14 bits, which has the effect of removing the fractional part of the fixed-point value of the computed fog value (see the Appendix for information on fixed-point numbers). Finally, we clamp the computed fog value so that it lies within the range of 0 to MAX_LIGHT_LEVELS, and call the `fog_native` method of the `sinfo` object to apply the fog to the pixel.

Mesa/OpenGL Z Buffering

Mesa/OpenGL provide built-in support for *z* buffering. 3D acceleration hardware often provides a *z* buffer in the hardware, meaning that applications can reap the benefits of the *z* buffer (draw polygons in any order, the display will always be correct) without worrying too much about the per-pixel performance hit; in hardware, the calculations are fast enough so as not to be the bottleneck. This does not mean, however, that higher-level culling (such as hierarchical view frustum culling) can be completely ignored; it is still a waste of time, even on the fastest accelerated hardware, to send millions of polygons to be drawn by the hardware if 99% of them are obscured or

invisible. In other words, hardware *z* buffering allows you to focus on high-level culling, but it does not solve all VSD problems by itself.

Performing *z* buffering through Mesa/OpenGL is done by issuing OpenGL commands to enable *z* buffering. OpenGL calls the *z* buffer the "depth buffer." The following new classes call the appropriate OpenGL routines to enable *z* buffering.

- `l3d_screen_zbuf_mesa`: A screen class identical to `l3d_screen_mesa`, except that we add the parameter GLUT_DEPTH to the call to `glutInitDisplayMode`, to tell GLUT to use the depth buffer.
- `l3d_rasterizer_3d_zbuf_mesa_imp`: A rasterizer implementation identical to the 3D Mesa rasterizer implementation, except that the constructor calls `glEnable(GL_DEPTH_TEST)` to turn on depth buffering, and the `clear_buffer` method calls `glClear(GL_DEPTH_BUFFER_BIT)` to also clear the depth buffer in addition to the frame buffer.

With these simple changes, our Mesa/OpenGL rasterizer now performs *z* buffering.

Note, however, that the depth buffering only works because the original `ogl_texturing_between_glbegin_glend` method passed the pre-projection constant of 1 as the *z* coordinate to the `glVertex4fv` function. This, combined with passing *z* as the *w* coordinate, led to the post-projection value of $1/z$ in the *z* component of the vertex. We commented on this in Chapter 2. This $1/z$ value then encodes the depth information and can be used for determining visibility among pixels from different polygons. If, on the other hand, we had passed the original *z* coordinate as both the *z* and *w* components to `glVertex4fv`, then after the homogeneous division by *w*, the *z* component would become the constant 1—smashed down into a plane, irrevocably losing the depth information. Try changing the code in `l3d_rasterizer_3d_mesa_imp` to pass the `orig_z` value as the third (*z*) component to `glVertex4fv`; you will then see (in the next sample program) that the *z* buffering no longer works correctly.

Listing 4-15: `sc_zmesa.h`

```
#ifndef __SC_ZMESA_H
#define __SC_ZMESA_H
#include "../tool_os/memman.h"

#include "sc_mesa.h"

class l3d_screen_zbuf_mesa : public l3d_screen {
  public:
    l3d_screen_zbuf_mesa(int xsize, int ysize);
    void open_screen(void);
    void blit_screen(void);
    virtual ~l3d_screen_zbuf_mesa(void);
};

class l3d_screen_factory_zbuf_mesa : public l3d_screen_factory {
  public:
    l3d_screen *create(int xsize, int ysize) {
      return new l3d_screen_zbuf_mesa(xsize,ysize);
    }
};

#endif
```

Listing 4-16: sc_zmesa.cc

```
#include "sc_zmesa.h"
#include "../tool_os/memman.h"

l3d_screen_zbuf_mesa::l3d_screen_zbuf_mesa(int xsize, int ysize) :
    l3d_screen(xsize,ysize)
{
  int argc=1;
  char *argv[1];
  argv[0] = "a";

  Visual *vis;
  int depth=0, bytespp=0, scanline_pad=0;
  Display *dpy;
  XPixmapFormatValues *pixmap_formats;
  int i, count;

  dpy = XopenDisplay(NULL);

  pixmap_formats = XListPixmapFormats(dpy, &count);
  for(i=0, depth=0; i<count; i++) {
    if(pixmap_formats[i].depth > depth) {
      depth        = pixmap_formats[i].depth;
      bytespp      = pixmap_formats[i].bits_per_pixel / BITS_PER_BYTE;
      scanline_pad = pixmap_formats[i].scanline_pad;
    }
  }
  Xfree(pixmap_formats);
  printf("max depth of display %d", depth);
  printf("bytes per pixel: %d", bytespp);

  bufsize = xsize * ysize * bytespp;

  vis = DefaultVisual(dpy,0);

  switch(vis->c_class) {

    case PseudoColor:
      sinfo = new l3d_screen_info_indexed_mesa(255, 65535, 65535, 65535);
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_DOUBLE|GLUT_INDEX|GLUT_DEPTH);
      glutInitWindowSize(xsize,ysize);
      glutInitWindowPosition(100,100);
      glutCreateWindow("l3d");

      glClearColor(0.0,0.0,0.0,0.0);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(0, xsize, ysize, 0);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      glTranslatef(0.375, 0.375, 0.0);
      glViewport(0,0,xsize,ysize);

      break;
    case TrueColor:
      sinfo = new l3d_screen_info_rgb_mesa(vis->red_mask, vis->green_mask,
                                           vis->blue_mask, bytespp);
```

```
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
      glutInitWindowSize(xsize,ysize);
      glutInitWindowPosition(100,100);
      glutCreateWindow("l3d");

      glClearColor(0.0,0.0,0.0,0.0);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(0, xsize, ysize, 0);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      glTranslatef(0.375, 0.375, 0.0);
      glViewport(0,0,xsize,ysize);

      break;
    case StaticColor: printf("unsupported visual StaticColor");break;
    case GrayScale:   printf("unsupported visual GrayScale");break;
    case StaticGray:  printf("unsupported visual StaticGray");break;
    case DirectColor: printf("unsupported visual DirectColor");break;
  }

}

l3d_screen_zbuf_mesa::~l3d_screen_zbuf_mesa(void) {
}

inline void l3d_screen_zbuf_mesa::blit_screen(void) {
  glutSwapBuffers();
  glFlush();
}

void l3d_screen_zbuf_mesa::open_screen(void) {
}
```

Listing 4-17: `ras3z_mes.h`

```
#ifndef __RAS3Z_MES_H
#define __RAS3Z_MES_H
#include "../tool_os/memman.h"

#include "ras3_mes.h"

class l3d_rasterizer_3d_zbuf_mesa_imp :
      public l3d_rasterizer_3d_mesa_imp
{
  public:
    l3d_rasterizer_3d_zbuf_mesa_imp(int xs, int ys, l3d_screen_info *si);
    virtual ~l3d_rasterizer_3d_zbuf_mesa_imp(void);
    void clear_buffer(void);
};

class l3d_rasterizer_3d_zbuf_mesa_imp_factory :
  public l3d_rasterizer_3d_imp_factory {
  public:
    l3d_rasterizer_3d_imp *create(int xs, int ys, l3d_screen_info *si);
};

#endif
```

Listing 4-18: ras3z_mes.cc

```
#include "ras3z_mes.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include "../tool_os/memman.h"

l3d_rasterizer_3d_imp * l3d_rasterizer_3d_zbuf_mesa_imp_factory::create
(int xs, int ys, l3d_screen_info *si)
{
  return new l3d_rasterizer_3d_zbuf_mesa_imp(xs,ys,si);
}

l3d_rasterizer_3d_zbuf_mesa_imp::
l3d_rasterizer_3d_zbuf_mesa_imp(int xs, int ys, l3d_screen_info *si):
    l3d_rasterizer_3d_imp(xs,ys,si),
    l3d_rasterizer_2d_imp(xs,ys,si),
    l3d_rasterizer_2d_mesa_imp(xs,ys,si),
    l3d_rasterizer_3d_mesa_imp(xs,ys,si)
{
  glEnable(GL_DEPTH_TEST);
}

l3d_rasterizer_3d_zbuf_mesa_imp::
~l3d_rasterizer_3d_zbuf_mesa_imp(void)
{
}

void l3d_rasterizer_3d_zbuf_mesa_imp::clear_buffer(void)
{
  l3d_rasterizer_2d_mesa_imp::clear_buffer();
  glClear(GL_DEPTH_BUFFER_BIT);
}
```

Factory Manager for Z Buffered Rasterizers

To allow l3d programs to use the *z* buffered versions of the rasterizers, we need to provide a new factory manager which allows for selection of factories that produce the new rasterizers. The `l3d_factory_manager_v_0_3` class does exactly this. It adds three new choices to the initial factory selection menu, which are the *z* buffered versions of the first three choices.

Listing 4-19: fact0_3.h

```
#ifndef __FACT0_3_H
#define __FACT0_3_H
#include "../tool_os/memman.h"

#include "fact0_2.h"

class l3d_factory_manager_v_0_3 : public l3d_factory_manager_v_0_2 {
  public:
    l3d_factory_manager_v_0_3(void);
    virtual ~l3d_factory_manager_v_0_3(void);
    int choose_factories(void);
};
 extern l3d_factory_manager_v_0_3 factory_manager_v_0_3;

#endif
```

Listing 4-20: fact0_3.cc

```
#include "fact0_3.h"

#include <stdio.h>

#include "../tool_os/dis_x11.h"
#include "../tool_os/dis_mesa.h"
#include "../tool_2d/sc_x11.h"
#include "../tool_2d/sc_x11sh.h"
#include "../tool_2d/sc_mesa.h"
#include "../raster/ras3_sw.h"
#include "../raster/ras3_mes.h"
#include "../dynamics/plugins/pl_linux.h"
#include "../raster/ras3z_sw.h"
#include "../raster/ras3z_mes.h"
#include "../tool_2d/sc_zmesa.h"
#include "../tool_os/memman.h"

l3d_factory_manager_v_0_3 factory_manager_v_0_3;

l3d_factory_manager_v_0_3::l3d_factory_manager_v_0_3(void) :
    l3d_factory_manager_v_0_2()
{
}

l3d_factory_manager_v_0_3::~l3d_factory_manager_v_0_3(void)
{
}

int l3d_factory_manager_v_0_3::choose_factories(void) {

  char input[80];
  int i;

  printf("which configuration?");
  printf("1. Software X11");
  printf("2. Software X11 with SHM extension");
  printf("3. Mesa (with 3DFX h/w accel if installed with Mesa)");
  printf("4. Software X11 with z buffer and fog");
  printf("5. Software X11 with SHM, z buffer, and fog");
  printf("6. Mesa with z buffer");
  gets(input);
  sscanf(input,"%d", &i);
  switch(i) {
    case 1:
      screen_factory = new l3d_screen_factory_x11;
      ras_2d_imp_factory = new l3d_rasterizer_2d_sw_imp_factory;
      dispatcher_factory = new l3d_dispatcher_factory_x11;
      ras_3d_imp_factory = new l3d_rasterizer_3d_sw_imp_factory;
      plugin_loader_factory = new l3d_plugin_loader_linux_factory;
      break;
    case 2:
      screen_factory = new l3d_screen_factory_x11_shm;
      ras_2d_imp_factory = new l3d_rasterizer_2d_sw_imp_factory;
      dispatcher_factory = new l3d_dispatcher_factory_x11;
      ras_3d_imp_factory = new l3d_rasterizer_3d_sw_imp_factory;
      plugin_loader_factory = new l3d_plugin_loader_linux_factory;
      break;
    case 3:
      screen_factory = new l3d_screen_factory_mesa;
```

```
      ras_2d_imp_factory = new l3d_rasterizer_2d_mesa_imp_factory;
      dispatcher_factory = new l3d_dispatcher_factory_mesa;
      ras_3d_imp_factory = new l3d_rasterizer_3d_mesa_imp_factory;
      plugin_loader_factory = new l3d_plugin_loader_linux_factory;
      break;
    case 4:
      screen_factory = new l3d_screen_factory_x11;
      ras_2d_imp_factory = new l3d_rasterizer_2d_sw_imp_factory;
      dispatcher_factory = new l3d_dispatcher_factory_x11;
      ras_3d_imp_factory = new l3d_rasterizer_3d_zbuf_sw_imp_factory;
      plugin_loader_factory = new l3d_plugin_loader_linux_factory;
      break;
    case 5:
      screen_factory = new l3d_screen_factory_x11_shm;
      ras_2d_imp_factory = new l3d_rasterizer_2d_sw_imp_factory;
      dispatcher_factory = new l3d_dispatcher_factory_x11;
      ras_3d_imp_factory = new l3d_rasterizer_3d_zbuf_sw_imp_factory;
      plugin_loader_factory = new l3d_plugin_loader_linux_factory;
      break;
    case 6:
      screen_factory = new l3d_screen_factory_zbuf_mesa;
      ras_2d_imp_factory = new l3d_rasterizer_2d_mesa_imp_factory;
      dispatcher_factory = new l3d_dispatcher_factory_mesa;
      ras_3d_imp_factory = new l3d_rasterizer_3d_zbuf_mesa_imp_factory;
      plugin_loader_factory = new l3d_plugin_loader_linux_factory;
      break;
  }

  factory_manager_v_0_2 = *this;
  factory_manager_v_0_2.should_delete_factories = 0;

  return i;
}
```

Sample Program: texzbuf

The sample program `texzbuf` is the same as the sample program `textest`, only the main program file has been changed to use the new factory manager. It displays two intersecting pyramids, one static and one rotating. Run the program, and select one of the factories 4, 5, or 6 to see the *z* buffering in action. Notice closely the area where the two pyramid objects intersect. With the previous program `textest` (and with this program `texzbuf`, as well, if you select factories 1, 2, or 3), the piercing polygons appear to flip immediately from in front of to behind the pierced polygons, due to the use of the simple painter's algorithm. With the new program `texzbuf`, using one of the new *z* buffer factories 4, 5, or 6, the area of intersection between the pyramid objects is correctly drawn. Notice that the tip of the one pyramid really appears to pierce the face of the other pyramid, and that there is a clear line of intersection between the pyramid objects, reflecting the real 3D positions of each pixel of each polygon. Again, this is one of the advantages of the *z* buffer algorithm: it is accurate to the pixel, even for intersecting polygons, and no polygons need to be split to produce a correct display.

Listing 4-21: `main.cc`, main program file for sample program `texzbuf`. Only the factory manager has been changed from previous program `textest`.

```
#include "../textest/textest.h"
#include "../lib/system/fact0_3.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdarg.h>

main() {
  l3d_dispatcher *d;
  l3d_pipeline_world *p;
  my_world *w;

  factory_manager_v_0_3.choose_factories();
  d = factory_manager_v_0_3.dispatcher_factory->create();

  w = new my_world();
  p  = new l3d_pipeline_world(w);
  d->pipeline = p;
  d->event_source = w->screen;

  d->start();

  delete p;
  delete d;
  delete w;
}
```

TEAMFLY

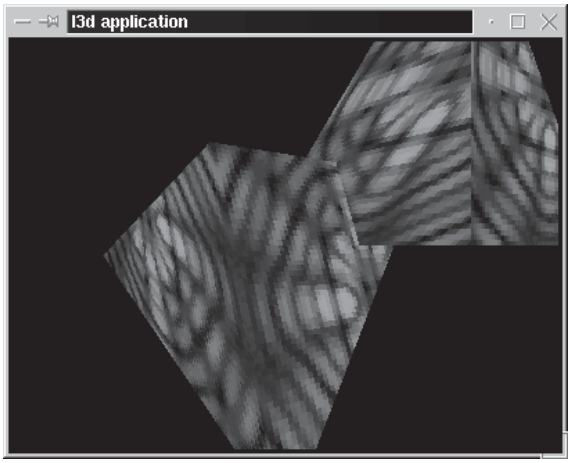


Figure 4-19: Output from sample program `texzbuf`. Notice that the polygons piercing the right pyramid appear partially in front of and partially behind the pierced polygons, just as they should.

Z Buffer-like Algorithms

It's worth mentioning that there are a number of variations on the *z* buffer algorithm which can also be useful. Initially, we viewed polygon rasterization in terms of a vertical collection of horizontal spans of pixels drawn to the screen. For each scanline in the polygon, we draw one horizontal span. Then, with the introduction of the *z* buffer algorithm, we shifted the focus to

Team-Fly®

individual pixels. Each individual pixel is tested against and possibly written to the *z* buffer. A solution between these two extremes is to keep track of spans of pixels for each scanline.

Instead of storing the *z* value for every pixel in each scanline, we store a list of spans for each scanline. Each span represents a visible, horizontally contiguous set of pixels belonging to one polygon, along with depth information just for each of its endpoints. This is the difference between spans and the *z* buffer: the *z* buffer explicitly stores the depth information for every pixel within (and even outside of) the span, whereas the span-based method just stores the depth information for the endpoints of the span. Since by definition the span is a horizontally contiguous set of pixels belonging to one polygon, no information is lost by just storing the depth values at the endpoints. An intersection routine which splits spans when they overlap replaces the pixel-for-pixel comparison of the *z* buffer.

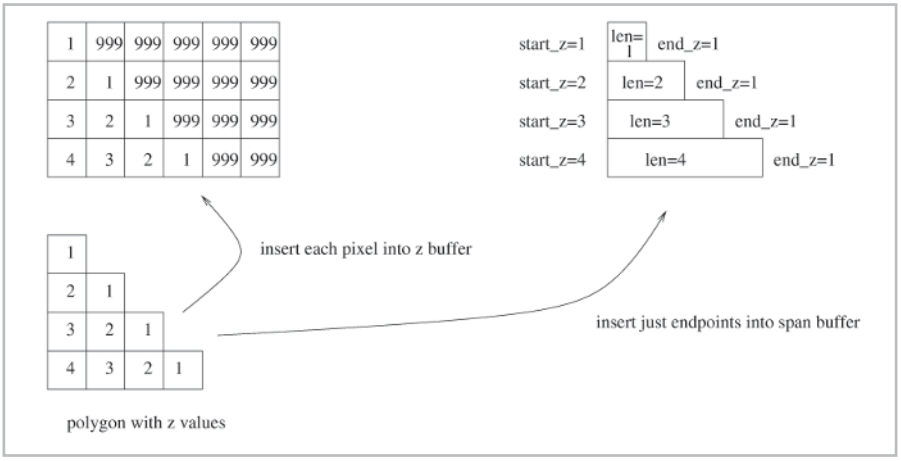


Figure 4-20: z buffering versus span buffering. Instead of storing depth values for each pixel within a span (left), we can store just the starting and ending values, and the span length (right). This saves memory, but overlapping spans require an intersection computation and creation of new spans.

Consider one scanline. Initially, its span list is empty. Then, if we rasterize a horizontal span of a polygon into the scanline, we insert this span—consisting of its left and right *x* and *z* values—into the span list. Later, as we rasterize other polygons into the same scanline, we insert their spans into the same list. If two spans do not overlap—in other words, if their *x* intervals are disjoint—we do not need to waste time checking pixel depths; we know that the two spans cannot obscure each other. But if two spans overlap, then we split the spans by analytically computing which parts of both spans are visible; this is nothing more than an intersection calculation to see at exactly which *x* value one span "pokes through" (has a *z* value greater than) the other span. This analytical computation is possible because we store the depth information with each endpoint of each span. So, instead of individually checking each pixel against the *z* buffer, we group pixels into spans. When spans overlap, we compute an analytical intersection and split the spans.

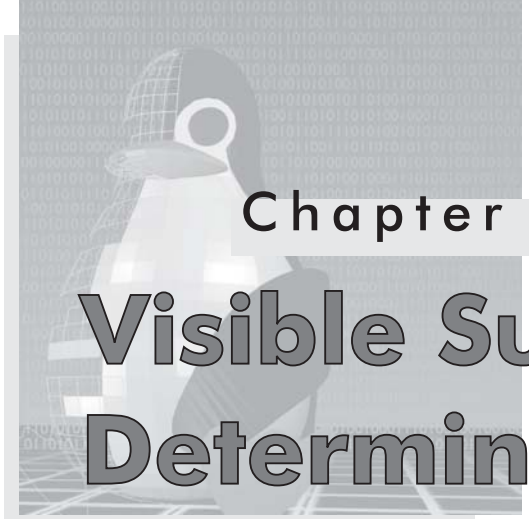
Another span-based VSD algorithm requires no span splitting, but also requires us to sort the polygons from front to back. We draw the frontmost polygons first. As each polygon is drawn, we write a span into the scanline. This span indicates the portion of the scanline which has already been covered by one of the frontmost polygons. We continue drawing polygons, proceeding towards the backmost polygons. If two spans overlap, we merge the two into one larger span. We never redraw a pixel in a location where a span already exists. The spans in this case thus merely keep track of coverage information, and not depth information. This is sort of a "reverse painter's algorithm," where we only draw background polygons around the areas already covered by foreground polygons. This algorithm requires perfect front-to-back sorting, and does not handle intersecting polygons. However, it can provide a useful measure of when a scene has been completely rendered. If every span for every scanline is completely full, then we know that every pixel in the display has been rendered. This completion test requires only one check per scanline, as opposed to one check per pixel with a *z* buffer. Such a completion test can be useful for some front-to-back VSD algorithms which continue drawing polygons until the entire screen has been filled.

Span-based algorithms can provide a speed-up over the pixel-based *z* buffer because they work with groups of pixels instead of individual pixels. The problem is that using spans requires a number of per-scanline data structures which do not map well onto 3D acceleration hardware, where pixel-level or scanline-level access is either not possible or comparatively slow.

Summary

In this chapter we took a look at some important and generally applicable visible surface determination algorithms. The goals of VSD are correctness and efficiency. We looked at back-face culling, hierarchical view frustum culling and bounding volumes, the painter's algorithm, and *z* buffering.

Chapter 5 continues the discussion of VSD with some specific algorithms based on space-partitioning techniques.



Chapter 5

Visible Surface Determination II: Space-partitioning Techniques

This chapter continues the discussion of various VSD algorithms that was started in Chapter 4. Chapter 4 dealt with generally applicable VSD techniques. This chapter deals with space-partitioning techniques, which represent a "divide and conquer" approach to VSD. We partition, or divide, space into various regions, then perform VSD based on the regions. Of course, it is not all quite as simple as it sounds, which is why we take the time to look at these algorithms in detail in this chapter.

This chapter covers:

- Binary space partitioning trees
- Octrees
- Regular spatial partitioning
- Portals and cells
- Special effects with portals (shadows, mirrors)
- Overview of other VSD algorithms.

Binary Space Partitioning Trees, Octrees, and Regular Spatial Partitioning

We mentioned in Chapter 4 the correctness and visibility aspects of VSD algorithms. The binary space partitioning (BSP) tree and octree algorithms are VSD algorithms which address both the correctness and efficiency issues of VSD. First, let's cover BSP trees; later, we'll look at octrees. The two algorithms are quite similar.

The BSP tree divides space hierarchically into regions in a time-consuming, one-time preprocessing step, then later allows determination of a correct back-to-front or front-to-back polygon ordering, for any viewpoint, without sorting the polygons (in contrast to the simple painter's algorithm). Since the construction of the BSP tree takes a lot of time, the BSP tree algorithm is generally best suited for scenes where most of the polygons do not move because moving polygons would force a recomputation of at least part of the tree. (There are ways of incorporating movement into BSP trees, but they are a bit tricky; see Bruce Naylor's article "A Tutorial on Binary Space Partitioning Trees" for details on so-called "tree merging." [NAYL98])

The BSP tree is based upon a very simple idea. Consider a space. Concretely, you can think of this "space" as a huge box large enough to contain all geometry in your virtual world. Then, consider a plane with arbitrary orientation located somewhere within this space. As we have seen, a plane is infinitely wide and high, has a front side, and has a back side. Thus, we can think of the plane as dividing the space into two regions: the region in front of the plane and the region behind the plane. (We can also consider the region exactly on the surface of the plane to be a third region, or we can assign it arbitrarily to the front or back side.)

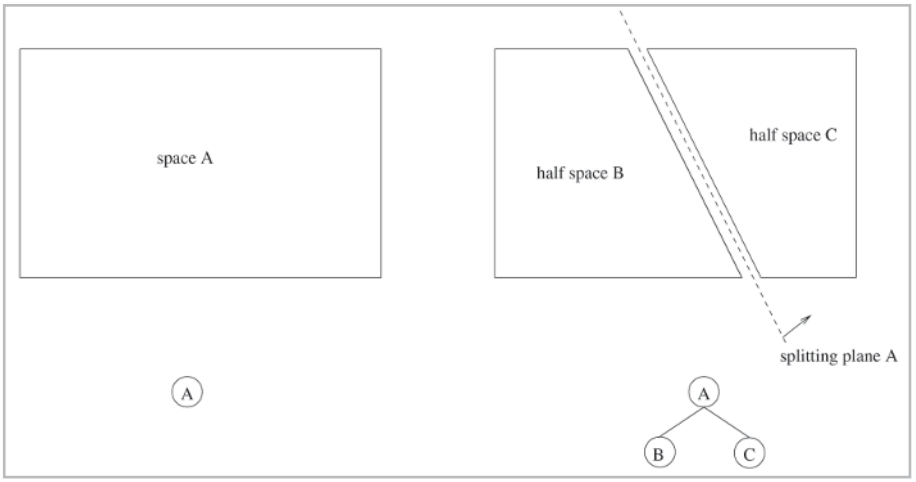


Figure 5-1: A plane divides space into two regions—three, if we count the surface of the plane itself.

We use the term *half space* to describe each region that results from the splitting of a space by a plane. Then, given a half space, we can again choose a splitting plane within this half space, which again results in two half spaces. We can continue this splitting process indefinitely.

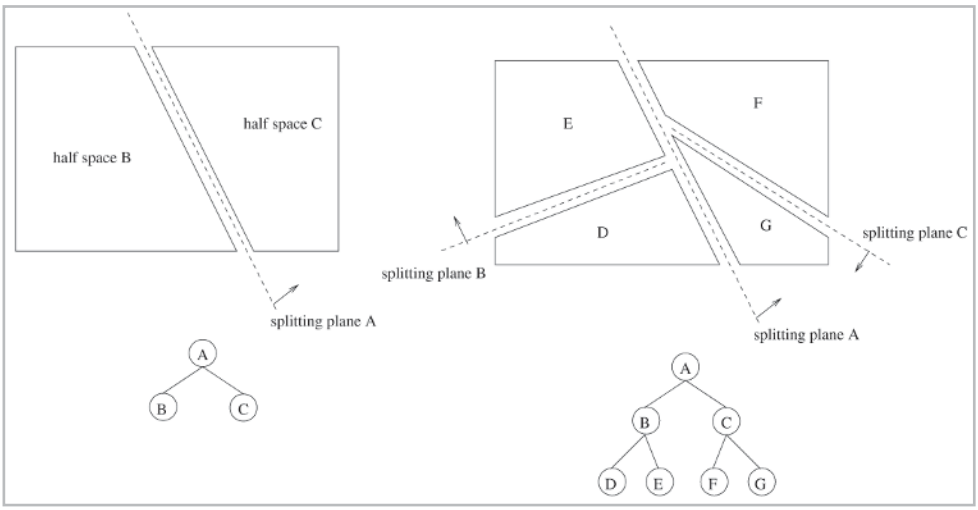


Figure 5-2: Each half space can be split again by a plane.

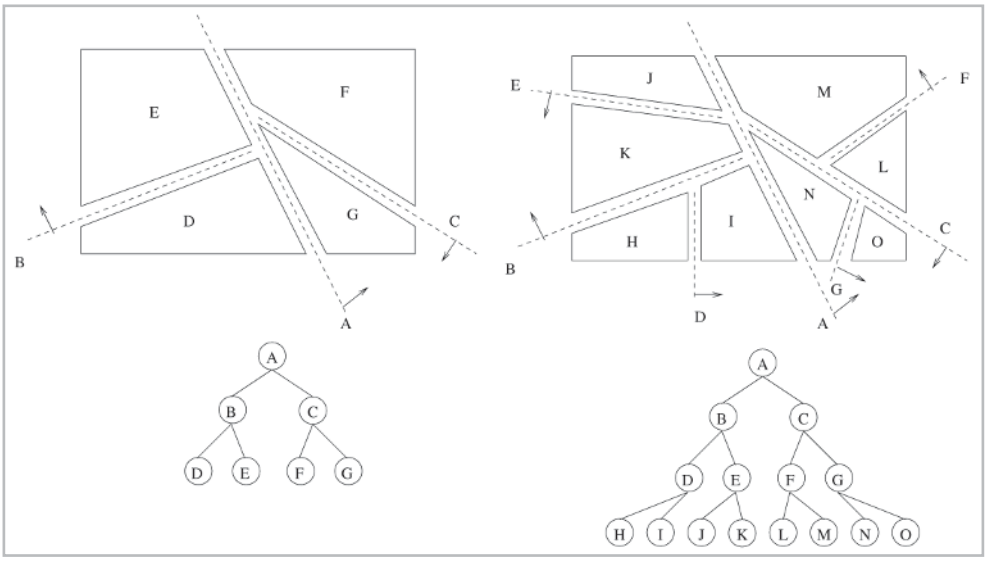


Figure 5-3: The process of half space splitting can be extended to an arbitrary number of levels.

Thus, we are using planes to hierarchically slice space into smaller and smaller regions. We can store such a hierarchical space division in memory as a tree data structure. Each internal node of the tree contains a plane, and has two child nodes: one for the half space on the front side and one for the half space on the back side of the plane. Each node of the tree also represents the small region of space which remains after being sliced by all parent nodes along the path to the node. This tree is the BSP tree: it is a binary partition of space. It is binary because a plane has two sides, a front and a back; it is a space partition, because we use the plane to split or partition the space into

exactly two pieces. Let us use the convention that the left child of a BSP tree node represents the space behind the partitioning plane of the parent, and the right child is the space in front of the partitioning plane.

Notice that we thought of the original space as a huge bounding box enclosing all of the geometry in our world. For any size of geometry, it is always possible to make a bigger bounding box to enclose all of it. Thus, we can always think of the entire world space as being a huge box. A box is convex; it has no concave dents or holes in it. And when repeatedly slicing a convex shape with a plane, the resulting pieces are also always convex. This is because each plane is infinitely wide and high, and slices all geometry along its way, within the space or half space it divides. To understand this geometrically, take a piece of paper and a pair of scissors. The paper represents the original space. Cut the paper in two, using only one straight cut. Notice that each piece is still convex. Continue cutting each half piece in two, again using only straight cuts that go all the way from one side of the paper to the other, forming two pieces. You'll notice that you can never produce a concave piece by slicing in this manner; this is because each slice goes straight through the entire piece, preserving convexity.

The question remains as to why we want to partition space in this manner. Consider a set of polygons located within a space. If we can partition the polygons into two disjointed spaces, called A and B, then this partitioning encodes a viewpoint-independent visibility relationship between A and B. Note that if any polygons cross the boundary between A and B, they must be split into two parts, one completely in A and one completely in B. Essentially, the partition states "A is separate from B." This is also the reason that polygons must be split if they cross the partition boundary: the regions must be separate. Then, given this separation of regions, we can use the camera position combined with the partitioning to determine the relative visibility of A and B. If the camera is in A, then no polygon in B can obscure any polygon in A. Similarly, if the camera is in B, then no polygon in A can obscure any polygon in B [SCHU69]. The partitioning, combined with the camera position, gives us a visibility ordering among the regions of space; using the painter's algorithm, we can draw the far space first, then the near space. Then, within each smaller half space, we again use the camera position with respect to the half space's partitioning plane to determine the ordering of the sub-half-spaces within each half space. We continue recursively all the way down to the polygon level (exactly how polygons fit into the scheme is covered in the next section), which gives us a visibility ordering for all polygons within the scene—without the need for sorting the polygons.

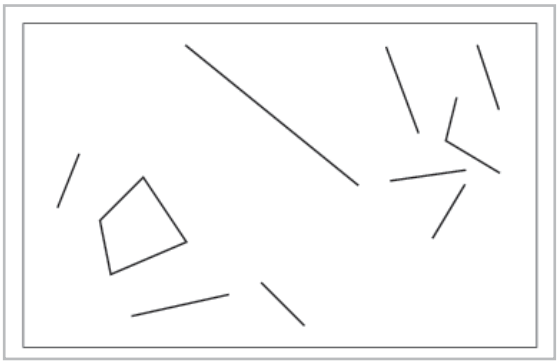


Figure 5-4: A set of polygons within a space.

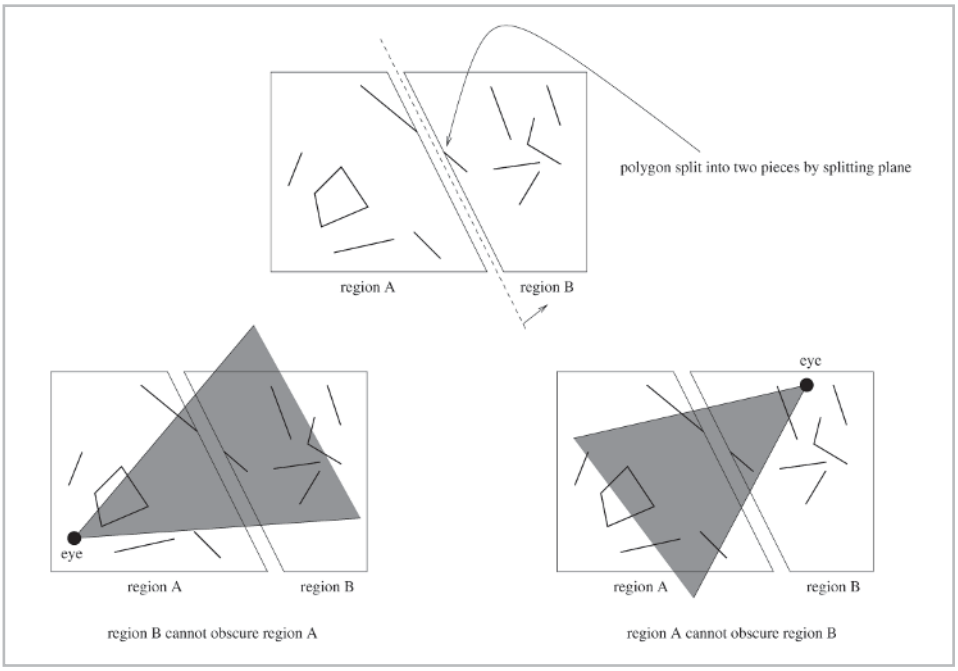


Figure 5-5: The space has been partitioned into two regions. If the camera is in region A, we know that no objects from region B can obscure any objects from region A. Similarly, if the camera is in region B, we know that no objects from region A can obscure any objects from region B. Note that one polygon, which crossed the splitting plane, had to be split into two pieces. If each smaller region is further subdivided, we apply the same visibility ordering principle recursively.

Using a BSP Tree to Partially Pre-sort Polygons

Let's now look more closely at how we can construct a BSP tree, then look at an example of how to use a BSP tree for rendering polygons in front-to-back or back-to-front order.

The key idea of the BSP tree is to divide space hierarchically using planes. The question is, which planes do we choose to divide space? The answer depends on the fact that the spatial regions the BSP tree partitions are not empty. Instead, these regions contain polygons. We typically choose one of the polygons within the space, and use the plane of the polygon to split the space into two smaller sub-spaces. If any of the remaining polygons cross the splitting plane, they must be split into two pieces, one in each sub-space, to ensure complete separation among the sub-spaces. Then, for each sub-space, we again choose a polygon within the sub-space, whose plane then splits all of the polygons within the sub-space. The splitting process continues recursively. We stop when all polygons have been used in this manner—in other words, when all polygons have been inserted into the tree.

The above discussion ignored one small detail; if several polygons are coplanar (share the same plane), then if any one of these polygons is chosen as the source of the splitting plane, we should ideally assign all of the coplanar polygons to the BSP tree node. Not doing so won't cause

any problems, but it causes inefficiency in the tree because later on we might choose a different one of the coplanar polygons, causing several nodes in the BSP tree to have the same splitting plane.

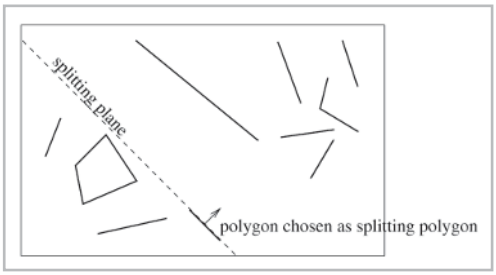


Figure 5-6: We usually choose the splitting plane from one of the polygons within the half space.

Choosing a Splitting Plane

The question still remains as to which polygon's plane, at any given point, is the best plane to choose for splitting the space. The answer depends on how we use the BSP tree. But generally, it is best to try to choose the splitting planes such that, after splitting, larger sub-spaces contain few polygons and smaller sub-spaces contain many polygons [NAYL98]. Striving for a balanced tree is usually not desirable, because the underlying geometry is usually not evenly distributed [NAYL98]. Remember that each node in the BSP tree represents a convex region formed by slicing up space by all planes along the path to the node. Furthermore, the region of a node is always large enough to contain all of the geometry beneath it in the BSP tree. By choosing the splitting planes as described above (many polygons in small spaces), the resulting BSP tree structure places volumes containing many polygons at relatively shallow levels of the tree. Assuming such groups of polygons represent local and not global features of interest in the underlying polygonal geometry (Bruce Naylor refers to this as the *principle of locality* [NAYL98]), then such a tree allows us to find volumes containing features of interest quickly, because we don't need to traverse the tree too deeply to find the volumes containing interesting groups of polygons representing local features of the geometry.

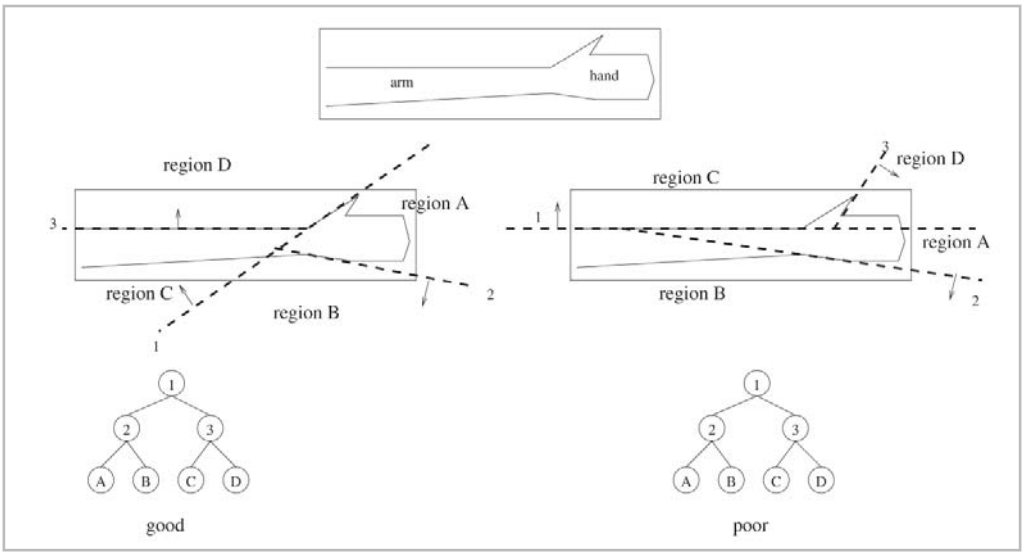


Figure 5-7: An arm/hand polygonal model and two different BSP trees for the same model. All vertex normals for the polygons in the hand model point outward; the splitting planes' normals point in the same direction as the corresponding polygons' normals. The BSP tree is constructed with the convention that the left child lies behind the splitting plane; the right child, in front of the plane. Note that for simplicity the BSP trees shown here are not complete, since a complete tree would require using every polygon as a splitting plane.

To understand this partitioning principle, consider the model in Figure 5-7. In this model, we have many polygons around the hand and fingers, and few polygons around the arm. The principle of locality states that the group of many polygons probably represents an interesting set of local features. Indeed it does—it represents the "hand" of our model. With a BSP tree, it is better to partition space such that the hand, containing many polygons, is in one small region, and the arm, containing few polygons, is in one large region. This is the BSP tree illustrated in the bottom left of the figure. An alternative, more balanced but poorer tree, is at the bottom right of the figure. Notice that the first splitting plane in the poorer, balanced tree splits the interesting region (the hand) into two regions.

Then, consider the question of point classification—is a particular point in the hand, or not? For instance, if you were simulating a hand catching a ball, the center point of the ball would need to be in the region of the hand in order to be caught. With the good tree on the left, as soon as we find out that the point is in the spatial region of node 3, which is the union of the children regions C and D, we know that the point cannot lie within the hand. (We classify a point in this manner by evaluating the plane equation of the splitting plane with the point's coordinates, as discussed in the introductory companion book *Linux 3D Graphics Programming*.) Thus, with the good tree, we can potentially know the answer to the point classification question at the first level of the BSP tree. Otherwise, if the point is in region 2, we further traverse this primarily hand-containing branch of the tree to compare the point with smaller and smaller spaces (defined by their splitting planes) until we finally know if the point is in the hand or not.

With the poor tree on the right, only region B can be excluded with certainty from the hand; and reaching region B requires us to traverse deeper in the tree to the second level. Furthermore, the interesting hand region has been split into the three regions A, C, and D, located in different sub-trees. Contrast this with the good BSP tree, where the interesting region was all in one sub-tree, adhering to the principle of locality. The split of the interesting region into many smaller parts means that to answer the question of point classification, we have to traverse deeper into the tree in many different branches. By using the principle of locality and trying to group large numbers of polygons into small areas, we try to make interesting regions reachable quickly through the tree for quicker point classification.

Naylor presents algorithms using expected case heuristics (rules of thumb guiding the algorithm at each stage to make a "best guess") to choose the splitting plane that most likely leads to good trees, where "good" is specified as a probability function [NAYL98]. Producing good trees requires the use of such guiding heuristics because the only known way of producing an optimal tree is by exhaustive search [NAYL98], which is far too slow.

Another possibility for choosing a splitting plane is to use the number of polygon splits as the criteria for choosing a splitting plane. We choose a few splitting planes at random, then see how many polygons would need to be split for each chosen splitting plane. The plane causing the fewest splits is then chosen [FOLE92]. However, minimizing splits does not necessarily automatically maximize the principle of locality. It is possible to construct scenes where no polygon's plane would split any other; then, if minimizing splits is the only guiding heuristic, plane selection becomes arbitrary. Combining several heuristics is generally the best way to construct good trees.

Back-to-Front Rendering (Painter's Algorithm Revisited)

Let's now see how we can use the BSP tree, once constructed. It should be noted that there are many uses of BSP trees; we can only cover a few here.

One interesting property of a BSP tree as constructed above (choose splitting planes based on polygons) is that a traversal of the tree by using the camera position produces a perfect back-to-front or front-to-back ordering of the polygons, just as needed for the painter's algorithm, but without sorting. The traversal, which is recursive, works as follows.

1. Set the current node to be the root of the BSP tree.
2. If the current node is a leaf node, then draw the polygon within the node and return to the caller.
3. Otherwise, compare the camera position to the splitting plane of the current node.
 - 3a. If the camera position is in front of the plane, then recursively draw the left sub-tree (behind the plane), then draw the polygon in the current node (on the plane), then recursively draw the right sub-tree (in front of the plane).
 - 3b. If, on the other hand, the camera position is behind the plane, recursively draw the right sub-tree (in front of the plane, but behind the plane from the camera's point of view), then draw the polygon in the current node (on the plane), then recursively draw the left sub-tree (behind the plane, but in front of the plane from the camera's point of view).

Team-Fly®

Notice that this is an in-order traversal of the tree data structure, with the modification that at each step of the recursive traversal, the position of the camera in respect to the splitting plane determines the order in which the children are visited. We always draw the sub-tree farthest from the camera first, then the polygon on the splitting plane, and then the sub-tree nearest the camera.

Let's restate that one more time for emphasis: the BSP traversal results in a perfect back-to-front ordering of polygons. We can draw polygons during the traversal, in a manner similar to the painter's algorithm, with the knowledge that nearer polygons then obscure farther ones. Also, because the space partitioning splits polygons which cross region boundaries, and because no farther region can ever obscure any other nearer region, the BSP traversal does not suffer from the problems of non-orderable polygon groups or intersecting polygons which we encountered in the simple painter's algorithm. Polygon ambiguities are resolved beforehand by the BSP tree by splitting the polygons across sub-spaces.

Front-to-Back Rendering

Instead of traversing from back to front, we can also traverse the BSP tree front to back. This requires a trivial change to the traversal logic: we simply recursively draw the nearest sub-tree first, then the polygon of the splitting plane, then the farthest sub-tree.

Front-to-back rendering, of course, requires some additional data structure, such as a *z* buffer or one of its span-based variants, to ensure that later pixels drawn from farther polygons do not obscure the nearer pixels already drawn. By eliminating recomputation of pixels that have already been drawn, a front-to-back BSP rendering can increase performance by a factor of two to four [GORD91].

Combining BSP Trees and Bounding Volumes

We've now seen how to use the BSP tree to produce a correct polygon ordering for an arbitrary camera position. But with very large scenes, it becomes impractical to draw all of the polygons in all of the nodes of the BSP tree. It turns out that the technique of hierarchical view volume culling works well with BSP trees.

The idea is that each node of a BSP tree represents a convex volume of space. So, all we need to do is compute a bounding sphere for each node in the BSP tree. This is the bounding sphere of the object formed by all of the polygons located at or beneath the node in the tree. We can then store this bounding sphere with each node of the tree.

During traversal of the BSP tree, we simply check to see if the bounding sphere for the current node is outside the view frustum. If it is, then we immediately stop processing that branch of the tree: nothing beneath it can possibly be visible. This allows us to quickly eliminate large portions of the scene from further processing. We can also check if a sphere is completely inside the frustum; if so, then all of its geometry must be inside. The idea is exactly the same as the hierarchical view frustum culling which we discussed earlier; the new trick is that the BSP tree automatically hierarchically organizes polygons into convex volumes, which we can then reuse for view frustum culling.

Note that the use of the bounding sphere for culling at each level of the BSP tree reinforces the idea that a balanced tree is usually not better, and that adhering to the principle of locality is more

desirable. View frustum culling can be viewed as a search operation among polygons to find those which lie outside of the frustum. By balancing the BSP tree, we force the search to progress with equal probability throughout the entire tree. By grouping polygons into localized regions, we allow the search to progress in a region-based manner. Then, culling a region saves more work than with a balanced tree: small regions contain many polygons, and small regions are more likely cullable. Again consider the hand and arm model; assume the arm has 10 polygons, and the hand has 5,000. By grouping the many hand polygons into a spatially small region, as soon as we find out that the hand's small bounding sphere is outside the frustum, we can cull 5,000 polygons from processing. If, however, we balance the tree and split the arm and hand polygons into two large, balanced groups of 2,505 polygons each, then we cannot cull "the hand" as a whole; instead, we have to look at the left arm-hand half (5+2,500 polygons) and the right arm-hand half (5+2,500) polygons separately, neither half of which is likely to be culled because the large volumes contain relatively many polygons, and larger volumes are difficult to cull. This forces us to progress further and further down into the (balanced) tree to find which polygons are inside and which are outside the frustum.

Sample Program: bsp

With this theoretical understanding of BSP trees, let's take a look at a simple sample program that creates a BSP tree and renders it back to front, combining this with hierarchical view frustum culling. The program reads a single Videoscape mesh from disk and creates a BSP tree from these polygons, also storing bounding spheres with each node of the tree. Then, the tree is traversed and rendered in back-to-front order, using view frustum culling to cull parts of the tree. The program displays how many nodes of the tree have been visited for the current scene. Navigate around the scene and notice how the number changes; without view frustum culling, we would always need to visit all nodes of the tree.

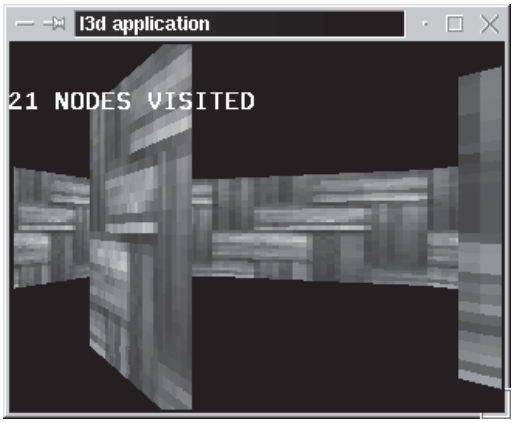


Figure 5-8: Output from sample program `bsp`.

The purpose of the next sample program `bsp` and the supporting classes is merely to illustrate the creation and use of 3D BSP trees. The sample program is not very efficient in terms of memory usage, so as it stands, it should not be used for creating BSP trees on models with more than a handful of polygons (unless you have lots of time and memory).

Classes l3d_halfspace and l3d_bsptree

Class `l3d_bsptree` is the class representing the nodes of a BSP tree and the operations which can be performed on the tree. The individual nodes of the BSP tree are instances of class `l3d_halfspace`. Each `l3d_halfspace` represents a region of space, can store a 3D object (list of polygons) located within the half space, can store a splitting plane, and can have child nodes.

Listing 5-1: `halfspace.h`

```
#ifndef __BSPTREE_H
#define __BSPTREE_H
#include "../../tool_os/memman.h"

class l3d_halfspace;
class l3d_object_clippable_boundable;
class l3d_object;
class l3d_bsptree {
  protected:
    void _permanently_merge_objects(l3d_object *dest, l3d_object *src);
    void _recursive_delete_tree(l3d_halfspace *root);

  public:
    l3d_halfspace *root;
    l3d_bsptree(void);
    virtual ~l3d_bsptree(void);
    void compute_for_object(l3d_halfspace *root,
                            l3d_object_clippable_boundable *rest_of_obj);

    void push_polys_to_children
    (l3d_halfspace *tree,
     l3d_object_clippable_boundable *unsplit_obj_from_parent);

    void make_videoscape_meshes(l3d_halfspace *tree);
    void print_stats(l3d_halfspace *tree);
};

#endif
```

Listing 5-2: `bsptree.h`

```
#ifndef __BSPTREE_H
#define __BSPTREE_H
#include "../../tool_os/memman.h"

class l3d_halfspace;
class l3d_object_clippable_boundable;
class l3d_object;
class l3d_bsptree {
  protected:
    void _permanently_merge_objects(l3d_object *dest, l3d_object *src);
    void _recursive_delete_tree(l3d_halfspace *root);

  public:
    l3d_halfspace *root;
    l3d_bsptree(void);
    virtual ~l3d_bsptree(void);
    void compute_for_object(l3d_halfspace *root,
                            l3d_object_clippable_boundable *rest_of_obj);
```

```
    void push_polys_to_children
    (l3d_halfspace *tree,
     l3d_object_clippable_boundable *unsplit_obj_from_parent);

    void make_videoscape_meshes(l3d_halfspace *tree);
    void print_stats(l3d_halfspace *tree);
};

#endif
```

Listing 5-3: bsptree.cc

```
#include "bsptree.h"
#include "halfspace.h"
#include "../object/obound.h"
#include "../../dynamics/plugins/vidmesh/vidmesh.h"
#include "../../tool_os/memman.h"

l3d_bsptree::l3d_bsptree(void) {
  root = NULL;
}

void l3d_bsptree::_recursive_delete_tree(l3d_halfspace *root) {
  if(root) {

    _recursive_delete_tree(root->children[0]);
    _recursive_delete_tree(root->children[1]);
    if(root->object) {
      delete root->object;
    }
  }
}

l3d_bsptree::~l3d_bsptree(void) {
  _recursive_delete_tree(root);
}

void l3d_bsptree::print_stats(l3d_halfspace *tree) {
  static int level = 0;
  level++;
  if(tree) {
    printf("%-*.*sLEVEL %d ", level*2, level*2, "", level);
    if(tree->object) {
      printf("%d polys, obj %p", tree->object->polygons.num_items, tree->object);
    }else {
      printf("NO OBJECT (0 polys)");
    }
    printf("%-*.*sleft child:", level*2, level*2, "");
    print_stats(tree->children[0]);
    printf("%-*.*sright child:", level*2, level*2, "");
    print_stats(tree->children[1]);
  }
  level--;
}

void l3d_bsptree::compute_for_object(l3d_halfspace *root,
                                     l3d_object_clippable_boundable *rest_of_obj)
{
  static int recurselevel=0;
```

```
recurselevel++;

if(rest_of_obj->polygons.num_items == 1) {
  root->object = rest_of_obj;

  l3d_polygon_3d_clippable *pc;
  pc = dynamic_cast<l3d_polygon_3d_clippable *>
       ((rest_of_obj->polygons)[0]);

  root->plane = pc->plane;

  root->children.num_items = 0;
  int cnum;
  root->children[cnum=root->children.next_index()] = NULL;
  root->children[cnum=root->children.next_index()] = NULL;

}
else {
  rest_of_obj->reset();

  int split_poly_num = rand() % rest_of_obj->polygons.num_items;
  l3d_polygon_3d_clippable *split_poly;
  split_poly = dynamic_cast<l3d_polygon_3d_clippable *>
               ((rest_of_obj->polygons) [split_poly_num]);

  root->plane = split_poly->plane;

  l3d_object_clippable_boundable *left_obj, *right_obj;
  left_obj = new l3d_object_clippable_boundable(1);
  *left_obj = *rest_of_obj;
  right_obj = new l3d_object_clippable_boundable(1);
  *right_obj = *rest_of_obj;

  root->object = rest_of_obj;
  root->bounding_sphere.compute_around_object(root->object);

  root->object->reset();
  l3d_polygon_3d_node *n;
  n = root->object->nonculled_polygon_nodes;
  while(n) {
    l3d_polygon_3d_clippable *pc;
    pc = dynamic_cast<l3d_polygon_3d_clippable*>(n->polygon);

    if(pc==split_poly) {
      root->object->nonculled_polygon_nodes = n;
      n->next = NULL;
      n->prev = NULL;
      break;
    }
    n = n->next;
  }

  root->object->make_current_geometry_permanent();

  left_obj->reset();

  int poly_i;
  poly_i = 0;
  n = left_obj->nonculled_polygon_nodes;
  while(n) {
```

```
  l3d_polygon_3d_clippable *pc;
  pc = dynamic_cast<l3d_polygon_3d_clippable*>(n->polygon);

  if(poly_i == split_poly_num) {
    if(n->prev) {
      n->prev->next = n->next;
    }else {
      left_obj->nonculled_polygon_nodes = n->next;
    }
    if(n->next) {
      n->next->prev = n->prev;
    }
    break;
  }
  n = n->next;
  poly_i++;
}
right_obj->reset();
poly_i = 0;
n = right_obj->nonculled_polygon_nodes;
while(n) {
  l3d_polygon_3d_clippable *pc;
  pc = dynamic_cast<l3d_polygon_3d_clippable*>(n->polygon);

  if(poly_i==split_poly_num) {
    if(n->prev) {
      n->prev->next = n->next;
    }else {
      right_obj->nonculled_polygon_nodes = n->next;
    }
    if(n->next) {
      n->next->prev = n->prev;
    }
    break;
  }
  n = n->next;
  poly_i++;
}

l3d_plane other_plane;
other_plane.a = -split_poly->plane.a;
other_plane.b = -split_poly->plane.b;
other_plane.c = -split_poly->plane.c;
other_plane.d = -split_poly->plane.d;

left_obj->clip_to_plane(other_plane);
right_obj->clip_to_plane(split_poly->plane);

left_obj->make_current_geometry_permanent();
right_obj->make_current_geometry_permanent();

l3d_halfspace *left, *right;
if(left_obj->polygons.num_items) {
  left = new l3d_halfspace;
  left->object = left_obj;
  left->children.num_items = 0;
  left->bounding_sphere.compute_around_object(left_obj);

}else {
  left = NULL;
```

```
    }
    if(right_obj->polygons.num_items) {
      right = new l3d_halfspace;
      right->object = right_obj;
      right->children.num_items = 0;
      right->bounding_sphere.compute_around_object(right_obj);

    }else {
      right = NULL;
    }

    root->children.num_items = 0;
    int cnum;
    root->children[cnum=root->children.next_index()] = left;
    root->children[cnum=root->children.next_index()] = right;

    if(left) {compute_for_object(left, left_obj); }
    if(right) {compute_for_object(right, right_obj); }
  }

  recurselevel--;

}

void l3d_bsptree::_permanently_merge_objects
(l3d_object *dest, l3d_object *src)
{
  if(!src) return;

  dest->reset();

  l3d_polygon_3d_node *n;
  n = src->nonculled_polygon_nodes;
  while(n) {

    l3d_polygon_2d *poly;
    poly = n->polygon->clone(); //- warning: does not clone texture

    //- point texture in cloned polygon to the destination mesh's texture,
    //- otherwise the cloned polygon still points to the original mesh's tex,
    //- which is a problem when the original mesh is deleted
    l3d_polygon_3d_textured *ptex;
    ptex = dynamic_cast<l3d_polygon_3d_textured *> (poly);
    if(ptex) {
      l3d_plugin_videoscape_mesh *mesh =
        (l3d_plugin_videoscape_mesh *)dest->plugin_data;

      ptex->texture->tex_data = mesh->tex_data;
    }
    poly->ivertices->num_items = 0;
    int ivtx;
    for(ivtx=0; ivtx<poly->clip_ivertices->num_items; ivtx++) {
      int new_ivtx;
      int found_new_ivtx=0;
      for(new_ivtx=0;
          new_ivtx<dest->vertices->num_fixed_items &&
          ! found_new_ivtx;
          new_ivtx++)
      {
        if( (*dest->vertices)[new_ivtx].original.X_ ==
```

```
            (**poly->vlist)[(*poly->clip_ivertices)[ivtx].ivertex].original.X_
            &&  (*dest->vertices)[new_ivtx].original.Y_ ==
            (**poly->vlist)[(*poly->clip_ivertices)[ivtx].ivertex].original.Y_
            &&  (*dest->vertices)[new_ivtx].original.Z_ ==
            (**poly->vlist)[(*poly->clip_ivertices)[ivtx].ivertex].original.Z_
          )
        {
          found_new_ivtx = 1;
          int next_idx = poly->ivertices->next_index();
          (*poly->ivertices)[next_idx].ivertex = new_ivtx;
        }
      }
      if(!found_new_ivtx) {
        dest->vertices->next_varying_index();
        dest->vertices->num_varying_items --;
        dest->vertices->max_varying_items --;
        dest->vertices->num_fixed_items ++;
        (*dest->vertices)
        [dest->vertices->num_fixed_items - 1].original
        =
          (**poly->vlist)[(*poly->clip_ivertices)[ivtx].ivertex].original ;

        int next_idx = poly->ivertices->next_index();
        (*poly->ivertices)[next_idx].ivertex =
          dest->vertices->num_fixed_items - 1;
      }

    }

    poly->vlist = &dest->vertices;

    int pnum = dest->polygons.next_index();
    l3d_polygon_3d_clippable *poly3;
    poly3 = dynamic_cast<l3d_polygon_3d_clippable *> (poly);

    dest->polygons[pnum] = poly3;
    poly3->compute_center();
    poly3->compute_sfcnormal();
    poly3->plane.align_with_point_normal
    (poly3->center.original,
     normalized(poly3->sfcnormal.original - poly3->center.original));

    n = n->next;
  }

}

void l3d_bsptree::push_polys_to_children
(l3d_halfspace *tree,
 l3d_object_clippable_boundable *unsplit_obj_from_parent)
{
  static int lvl=0;
  lvl++;
  if(tree) {

    //- define front and back clipping planes

    l3d_plane front_clip_plane, back_clip_plane;
    back_clip_plane.a = -tree->plane.a;
    back_clip_plane.b = -tree->plane.b;
```

```
back_clip_plane.c = -tree->plane.c;
back_clip_plane.d = -tree->plane.d;
front_clip_plane.a = tree->plane.a;
front_clip_plane.b = tree->plane.b;
front_clip_plane.c = tree->plane.c;
front_clip_plane.d = tree->plane.d;

l3d_object_clippable_boundable *back_obj;
back_obj = new l3d_object_clippable_boundable
           (tree->object->vertices->num_fixed_items);
int should_delete_back_obj = 1;

*back_obj = *tree->object;

//- uncomment the following to cause all polys to end up in the
//- right children in the tree (be sure to comment out the similar
//- assignment below to front_obj->polygons.num_items = 0)
//-
back_obj->polygons.num_items = 0;
if(unsplit_obj_from_parent) {
  unsplit_obj_from_parent->reset();
  unsplit_obj_from_parent->clip_to_plane(back_clip_plane);
  _permanently_merge_objects(back_obj, unsplit_obj_from_parent);
}

if(back_obj->polygons.num_items > 0) {
  back_obj->reset();

  if(tree->children[0]) {
    push_polys_to_children(tree->children[0], back_obj);
  }else {
    //- there previously were no polys in the back half space, but
    //- now due to a poly pushed down from the parent, there are.
    //- so create a new left child.
    tree->children[0] = new l3d_halfspace;
    tree->children[0]->object = back_obj;
    tree->children[0]->children.num_items = 0;
    tree->children[0]->children[tree->children[0]->children.next_index()]
    = NULL;
    tree->children[0]->children[tree->children[0]->children.next_index()]
    = NULL;
    tree->children[0]->bounding_sphere.compute_around_object(back_obj);

    should_delete_back_obj = 0;
  }
}
else {
  if(tree->children[0]) {
    push_polys_to_children(tree->children[0], NULL);
  }
}

if(should_delete_back_obj) {delete back_obj; }

l3d_object_clippable_boundable *front_obj;
front_obj = new l3d_object_clippable_boundable
            (tree->object->vertices->num_fixed_items);
int should_delete_front_obj = 1;
```

```
    *front_obj = *tree->object;

    //- uncomment the following to cause all polys to end up in the
    //- left children in the tree (be sure to comment out the similar
    //- assignment above to back_obj->polygons.num_items = 0)
    //-
    //- front_obj->polygons.num_items = 0;
    if(unsplit_obj_from_parent) {
      unsplit_obj_from_parent->reset();
      unsplit_obj_from_parent->clip_to_plane(front_clip_plane);
      _permanently_merge_objects(front_obj, unsplit_obj_from_parent);
    }
    if(front_obj->polygons.num_items > 0) {
      front_obj->reset();
      if(tree->children[1]) {
        push_polys_to_children(tree->children[1], front_obj);
      }else {
        //- there previously were no polys in the front half space, but
        //- now due to a poly pushed down from the parent, there are.
        //- so create a new right child.
        tree->children[1] = new l3d_halfspace;
        tree->children[1]->object = front_obj;
        tree->children[1]->children.num_items = 0;
        tree->children[1]->children[tree->children[1]->children.next_index()]
        = NULL;
        tree->children[1]->children[tree->children[1]->children.next_index()]
        = NULL;
        tree->children[1]->bounding_sphere.compute_around_object(front_obj);

        should_delete_front_obj = 0;
      }
    }
    else {
      if(tree->children[1]) {
        push_polys_to_children(tree->children[1], NULL);
      }
    }
    if(should_delete_front_obj) {delete front_obj; }

    delete tree->object;
    tree->object = NULL;
  }
  lvl--;
}

void l3d_bsptree::make_videoscape_meshes(l3d_halfspace *tree) {
  static int level = 0;
  static int filecounter=0;
  level++;
  if(tree) {
    if(tree->object) {
      FILE *fp;
      char fname[512];
      sprintf(fname,"vid%d.obj", filecounter);
      fp = fopen(fname,"wb");
      fprintf(fp,"3DG1");
      fprintf(fp,"%d", tree->object->vertices->num_fixed_items);
      for(int i=0; i<tree->object->vertices->num_fixed_items; i++) {
        fprintf(fp, "%f %f %f",
                l3d_real_to_float( (*tree->object->vertices)[i].original.X_),
```

TEAM-FLY

```
            l3d_real_to_float( (*tree->object->vertices)[i].original.Z_),
            l3d_real_to_float( (*tree->object->vertices)[i].original.Y_));
      }
      for(int ip=0; ip<tree->object->polygons.num_items; ip++) {
        fprintf(fp,"%d ", (*tree->object->polygons[ip]->ivertices).num_items);
        for(int iv = (*tree->object->polygons[ip]->ivertices).num_items-1;
            iv >= 0;
            iv--)
        {
          fprintf(fp,"%d ",
                  (*tree->object->polygons[ip]->ivertices)[iv].ivertex);
        }
        fprintf(fp,"0xf0f0f0");
      }
      fclose(fp);
      filecounter++;
    }else {
    }
    make_videoscape_meshes(tree->children[0]);
    make_videoscape_meshes(tree->children[1]);
  }
  level--;
}
```

First, let's look at class `l3d_halfspace`. Member variable `plane` is the splitting plane for this half space. Member variable `children` is an array of child nodes. `children[0]` is the left child (behind the splitting plane); `children[1]` is the right child (in front of the splitting plane). The reason for using an array is to enable each tree node to have more than two children, as is the case with the octree data structure (covered later in this chapter). Member variable `object` is the 3D object (in other words, a list of polygons) associated with this node. For the BSP tree as covered so far, this object will contain exactly one polygon; for the leafy BSP tree covered later, this object can contain more polygons. Member variable `bounding_sphere` is the bounding sphere for all polygons in this node and all nodes beneath it in the tree; in other words, it is the bounding sphere for all geometry within the half space.

Next, let's look at class `l3d_bsptree`. Member variable `root` is the pointer to the root of the tree of `l3d_halfspace` nodes. Method `compute_for_object` computes the BSP tree based on the given arbitrary object and a root node of type `l3d_halfspace`. It computes the BSP tree by choosing a splitting plane, partitioning the polygons with respect to this splitting plane, and recursively computing smaller sub-trees for each smaller subset of polygons. Specifically, the BSP tree construction works as follows.

1. Compute the bounding sphere for all polygons within the current half space, and store it within the current tree node.
2. Choose one of the polygons belonging to the current half space. The chosen polygon's plane is the current splitting plane. Store the splitting plane and the polygon in the node.
3. If there are no remaining polygons in the half space, then return to the caller; every polygon in this half space has been assigned to a tree node. Otherwise, split the remaining polygons against the split plane. Divide the split polygons into two groups: the group in front of the plane and the group behind the plane.

4. For the group behind the plane, create a left child of the current tree node, and recursively call step 1. For the group in front of the plane, create a right child of the current tree node, and recursively call step 1.

The algorithm terminates when no polygons are left in the input set because each has been assigned to a node.

Note that for simplicity this routine ignores the small detail of coplanar polygons; as mentioned earlier, when choosing a splitting plane we actually should assign all polygons lying on this splitting plane to the current node. This requires a search through the rest of the polygons to see if any of them lie within the splitting plane (evaluating all of the polygon's vertices with the plane equation answers this question). Not doing this, as mentioned earlier, doesn't harm, it just makes for a less efficient tree.

Also, for reasons of simplicity, the polygon we choose for the splitting plane is chosen completely at random. As mentioned earlier, creating a better tree requires us to specify the operations of interest upon the tree (e.g., point classification), and typically means creating a tree based on expected case heuristics which exploits the principle of locality.

Returning to the class `l3d_bsptree`, the protected method `_recursive_delete_tree` recursively deletes the tree rooted at the node stored in member variable `root`. This routine is called from the destructor.

The method `print_stats` recursively prints some statistics about the BSP tree. It shows how many polygons are in each node, and the left and right children of the node. For the BSP trees constructed as described above, each node contains exactly one polygon. (This changes with leafy BSP trees, as we see later.)

For now, ignore the methods `push_polys_to_children`, `make_videoscape_meshes`, and `_permanently_merge_objects`. We cover these in the coming section on leafy BSP trees.

Class l3d_world_bsptree

Class `l3d_world_bsptree` is a demonstration world class illustrating the basics of drawing a BSP tree in back-to-front order, combined with view frustum culling. Provision is only made for one BSP tree. In a larger application, it can be impractical to store the entire geometry for a world in one BSP tree (although it can be done); doing so raises practical questions, such as to how to assign texture objects to different polygons in different parts of the tree, how to handle moving objects, and so forth. A different way of organizing geometry involves a manual grouping of polygons into *sectors*, and the creation of a so-called *mini BSP tree*, one for each sector. See the sections on mini BSP trees and portals for further discussion of this technique.

Listing 5-4: `w_bsp.h`

```
#ifndef __W_BSP_H
#define __W_BSP_H
#include "../../tool_os/memman.h"

#include "../../tool_os/dispatch.h"
#include "../../view/camera.h"
#include "../../tool_2d/screen.h"
#include "../object/object3d.h"
```

```
#include "world.h"
#include "../frustum/vfrust.h"

class l3d_bsptree;
class l3d_halfspace;
class l3d_world_bsptree : public l3d_world
{
  protected:
    int visited_nodes;

  public:
    l3d_world_bsptree(int xsize, int ysize) :
    l3d_world(xsize,ysize) {};
    virtual ~l3d_world_bsptree(void);
    void render_bsp_tree(l3d_halfspace *tree, l3d_point campos,
                         l3d_viewing_frustum &frustum);

    /* virtual */ void draw_all(void);
    l3d_bsptree *the_tree;
};

#endif
```

Listing 5-5: w_bsp.cc

```
#include "w_bsp.h"
#include <stdlib.h>
#include <string.h>

#include "../object/sector.h"
#include "../polygon/polygon.h"
#include "../../tool_2d/screen.h"
#include "../../tool_os/dispatch.h"
#include "../../raster/rasteriz.h"
#include "../../tool_2d/scrinfo.h"
#include "../../system/fact0_2.h"
#include "w_bsp.h"
#include "../bsp/halfspace.h"
#include "../bsp/bsptree.h"
#include "../frustum/vfrust.h"
#include "../../tool_os/memman.h"

l3d_world_bsptree::~l3d_world_bsptree() {
  if(the_tree) delete the_tree;
}

void l3d_world_bsptree::render_bsp_tree
(l3d_halfspace *tree, l3d_point campos, l3d_viewing_frustum &frustum)
{
  visited_nodes++;

  if(tree) {

    if( tree->plane.side_of_point(campos) > 0 ) {
      if(tree->children[0]
          && frustum.intersects_sphere(tree->children[0]->bounding_sphere)
        )
      {
        render_bsp_tree(tree->children[0],campos,frustum);
      }
```

```
  if(tree->object) {
    tree->object->reset();

    //- no world coord xforms allowed, bsp is static

    if(tree->object->clip_to_plane(frustum.top)
        && tree->object->clip_to_plane(frustum.bottom)
        && tree->object->clip_to_plane(frustum.left)
        && tree->object->clip_to_plane(frustum.right)
        && tree->object->clip_to_plane(frustum.near)
        && tree->object->clip_to_plane(frustum.far))
    {

      tree->object->camera_transform(camera->viewing_xform);

      tree->object->apply_perspective_projection
      (*camera, screen->xsize, screen->ysize);

      l3d_polygon_3d_node *n;
      n = tree->object->nonculled_polygon_nodes;
      while(n) {
        l3d_polygon_3d_clippable *poly =
          dynamic_cast<l3d_polygon_3d_clippable *>
          (n->polygon);

        if( poly->sfcnormal.transformed.Z_ -
            poly->center.transformed.Z_ < float_to_l3d_real(0.05) )
        {
          poly->draw(rasterizer);
        }
        n = n->next;
      }
    }
  }

  if(tree->children[1]
      && frustum.intersects_sphere(tree->children[1]->bounding_sphere)
    )
  {
    render_bsp_tree(tree->children[1],campos,frustum);
  }
}else {
  if(tree->children[1]
      && frustum.intersects_sphere(tree->children[1]->bounding_sphere)
    )
  {
    render_bsp_tree(tree->children[1],campos,frustum);
  }

  if(tree->object) {
    tree->object->reset();
    //- no world coord xforms allowed, bsp is static

    if(tree->object->clip_to_plane(frustum.top)
        && tree->object->clip_to_plane(frustum.bottom)
        && tree->object->clip_to_plane(frustum.left)
        && tree->object->clip_to_plane(frustum.right)
        && tree->object->clip_to_plane(frustum.near)
        && tree->object->clip_to_plane(frustum.far))
    {
```

```
          tree->object->camera_transform(camera->viewing_xform);
          tree->object->apply_perspective_projection
          (*camera, screen->xsize, screen->ysize);

          l3d_polygon_3d_node *n;
          n = tree->object->nonculled_polygon_nodes;
          while(n) {
            l3d_polygon_3d_clippable *poly =
              dynamic_cast<l3d_polygon_3d_clippable *>
              (n->polygon);

            {
              if( poly->sfcnormal.transformed.Z_ -
                  poly->center.transformed.Z_ < float_to_l3d_real(0.05) )
              {
                poly->draw(rasterizer);
              }
            }
            n = n->next;
          }
        }
      }
      if(tree->children[0]
          && frustum.intersects_sphere(tree->children[0]->bounding_sphere)
        )
      {
        render_bsp_tree(tree->children[0],campos,frustum);
      }
    }
  }

}

void l3d_world_bsptree::draw_all(void) {
  int iObj, iFacet;

  rasterizer->clear_buffer();

  l3d_viewing_frustum frustum;

  l3d_point top_left, top_right, bot_right, bot_left;
  const int XOFF=2,YOFF=2;

  top_left.set(l3d_mulrr(l3d_divrr(int_to_l3d_real(XOFF - screen->xsize/2),
                                   l3d_mulri(camera->fovx,screen->xsize)),
                         camera->near_z),
               l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->ysize/2 - YOFF),
                                   l3d_mulri(camera->fovy,screen->ysize)),
                         camera->near_z),
               int_to_l3d_real(camera->near_z),
               int_to_l3d_real(1));

  top_right.set(l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->xsize-XOFF - screen->xsize/2),
                                    l3d_mulri(camera->fovx,screen->xsize)),
                          camera->near_z),
                l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->ysize/2 - YOFF),
                                    l3d_mulri(camera->fovy,screen->ysize)),
                          camera->near_z),
                int_to_l3d_real(camera->near_z),
```

```
                int_to_l3d_real(1));

    bot_right.set(l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->xsize-XOFF - screen->xsize/2),
                                      l3d_mulri(camera->fovx,screen->xsize)),
                            camera->near_z),
                  l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->ysize/2 - screen->ysize+YOFF),
                                      l3d_mulri(camera->fovy,screen->ysize)),
                            camera->near_z),
                  camera->near_z,
                  int_to_l3d_real(1));

    bot_left.set(l3d_mulrr(l3d_divrr(int_to_l3d_real(XOFF - screen->xsize/2),
                                     l3d_mulri(camera->fovx,screen->xsize)),
                           camera->near_z),
                 l3d_mulrr(l3d_divrr(int_to_l3d_real(screen->ysize/2 - screen->ysize+YOFF),
                                     l3d_mulri(camera->fovy,screen->ysize)),
                           camera->near_z),
                 camera->near_z,
                 int_to_l3d_real(1));

    frustum.create_from_points
    (top_left, top_right, bot_right, bot_left,
     camera->near_z, camera->far_z,
     camera->inverse_viewing_xform);

    //- traverse the BSP tree

    l3d_point camera_world_pos = camera->VRP;

    visited_nodes = 0;
    render_bsp_tree(the_tree->root, camera_world_pos, frustum);

    char msg[256];
    sprintf(msg,"%d nodes visited", visited_nodes);
    rasterizer->draw_text(0,32,msg);
  }
```

Member variable `the_tree` points to the single BSP tree which will be rendered by the world. Overridden method `draw_all` then does the drawing. It creates a view frustum in world coordinates, then calls the recursive routine `render_bsp_tree`. Method `render_bsp_tree` then recursively traverses the tree in a back-to-front method as described earlier: first draw the far sub-tree (as seen from the camera position), then the polygon in the current node, then the near sub-tree. While recursively traversing the tree structure, we also check each node's bounding sphere against the view frustum. Since the bounding sphere for a node encompasses all of the geometry at or beneath that node in the tree, we do not process that node or any node beneath it if the node's bounding sphere is completely outside of the frustum.

The Main Program

The main program file for `bsp` is quite simple. It declares a subclass of `l3d_world_bsptree`, which performs some typical world initialization and loads a Videoscape mesh. The filename of the mesh it loads is `soup.obj`, so named because an arbitrary collection of polygons is often called a *polygon soup*. Then, in the last part of the world constructor, we create a BSP tree with exactly one node and no children, and then invoke the `compute_for_object` method to

compute the full BSP tree for the loaded Videoscape object. The drawing of the BSP tree takes place in the parent class `l3d_world_bsptree`, in the inherited method `draw_all`.

Listing 5-6: `main.cc`, main program file for sample program `bsp`

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../lib/geom/object/sector.h"
#include "../lib/geom/polygon/p3_flat.h"
#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rast3.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/geom/world/w_bsp.h"
#include "../lib/system/fact0_2.h"
#include "../lib/pipeline/pi_wor.h"
#include "../lib/dynamics/plugins/plugenv.h"
#include "../lib/geom/texture/tl_ppm.h"
#include "../lib/dynamics/plugins/vidmesh/vidmesh.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdarg.h>
#include "../lib/geom/bsp/halfspace.h"
#include "../lib/geom/bsp/bsptree.h"

class my_world:public l3d_world_bsptree {
  protected:
    l3d_texture_loader *texloader;
    l3d_surface_cache *scache;
  public:
    my_world(void);
    virtual ~my_world(void);
};

my_world::my_world(void)
    : l3d_world_bsptree(320,240)
{
  l3d_screen_info *si = screen->sinfo;

  camera->VRP.set(0,0,-50,0);
  camera->near_z = float_to_l3d_real(1.0);
  camera->far_z = int_to_l3d_real(50);

  //- for mesa rasterizer's reverse projection
  rasterizer_3d_imp->fovx = &(camera->fovx);
  rasterizer_3d_imp->fovy = &(camera->fovy);
  rasterizer_3d_imp->screen_xsize = &(screen->xsize);
  rasterizer_3d_imp->screen_ysize = &(screen->ysize);

  int i,j,k;

  texloader = new l3d_texture_loader_ppm(si);
  scache = new l3d_surface_cache(si);
```

```
  //- create a plugin object
  l3d_object_clippable_boundable *oc;

  oc =
    new l3d_object_clippable_boundable(10);
  //- max 10 fixed vertices, can be overridden by plug-in if desired
  //- by redefining the vertex list

  oc->plugin_loader =
    factory_manager_v_0_2.plugin_loader_factory->create();
  oc->plugin_loader->load("../lib/dynamics/plugins/vidmesh/vidmesh.so");
  oc->plugin_constructor =
    (void (*)(l3d_object *, void *))
    oc->plugin_loader->find_symbol("constructor");
  oc->plugin_update =
    (void (*)(l3d_object *))
    oc->plugin_loader->find_symbol("update");
  oc->plugin_destructor =
    (void (*)(l3d_object *))
    oc->plugin_loader->find_symbol("destructor");
  oc->plugin_copy_data =
    (void (*)(const l3d_object *, l3d_object *))
    oc->plugin_loader->find_symbol("copy_data");

  texloader = new l3d_texture_loader_ppm(si);
  scache = new l3d_surface_cache(si);

  char plugin_parms[4096];
  sprintf(plugin_parms, "%d %d %d  1 0 0 0 1 0 0 0 1 soup.obj soup.ppm soup.uv",
          0,0,0);

  l3d_plugin_environment *e =
     new l3d_plugin_environment
       (texloader, screen->sinfo, scache, (void *)plugin_parms);

  if(oc->plugin_constructor) {
    (*oc->plugin_constructor) (oc,e);
  }

  oc->bounding_sphere.compute_around_object(oc);

  l3d_bsptree *tree;

  tree = new l3d_bsptree;
  tree->root = new l3d_halfspace;
  tree->root->bounding_sphere = oc->bounding_sphere;
  tree->root->object = NULL;
  tree->root->children.num_items = 0;
  tree->root->children[tree->root->children.next_index()] = 0;
  tree->root->children[tree->root->children.next_index()] = 0;

  tree->compute_for_object(tree->root, oc);

  the_tree = tree;

  screen->refresh_palette();
}

my_world::~my_world(void) {
  delete texloader;
```

```
  delete scache;
}

main() {
  l3d_dispatcher *d;
  l3d_pipeline_world *p;
  my_world *w;

  factory_manager_v_0_2.choose_factories();
  d = factory_manager_v_0_2.dispatcher_factory->create();

  w = new my_world();
  p  = new l3d_pipeline_world(w);
  d->pipeline = p;
  d->event_source = w->screen;

  d->start();

  delete d;
  delete p;
  delete w;
}
```

The World Database, Revisited

The `l3d_world` class, as presented thus far (see Chapter 1 and the introductory companion book *Linux 3D Graphics Programming*), uses a very simple world database: merely a list of `l3d_object` objects, traversed sequentially. As the number of objects grows, sequentially traversing the world database takes more and more time.

With this sample program `bsp`, we have now seen one example of an alternative world database. Instead of blindly processing all polygons, we have used the BSP tree to spatially "sort" the polygons hierarchically. Drawing the world then amounts to a hierarchical traversal of a binary tree.



TIP Notice that both the original simple world database (a list of objects) and the BSP's database (a binary tree of polygons) are both fundamental data structures from the field of computer science. There are many other data structures—queues, hash tables, directed acyclic graphs, and so forth. All of these can be used within the context of interactive 3D graphics applications to create more efficient data structures for managing larger world databases. While most data structures books deal (and necessarily so) with operations on simple objects such as words, letters, numbers, or database records, the same principles can be applied with great success to storing and efficiently accessing 3D objects in a world database. The portal scheme covered later in this chapter uses a graph to store visibility information.

Leafy BSP Trees: Automatic Convex Partitioning of Space

The BSP tree structure as described so far stores polygons in both the internal and leaf nodes of the tree. By traversing the tree in a particular order using the camera position relative to the current node's splitting plane, we determine an ordering among the BSP tree nodes. We draw each single

polygon in each node as we traverse the tree to determine a viewpoint-dependent polygon ordering without sorting.

Another form of BSP tree stores polygons exclusively in the leaf nodes of the tree, not in the internal nodes of the tree. Such a tree is sometimes called a *leafy BSP tree*. *Quake* was one of the first games to use this method. The idea is simple. When splitting the set of polygons by a plane, we effectively created three nodes. The left child is the root of the tree storing all polygons behind the plane, the right child is the root of the tree storing all polygons in front of the plane, and the current node stores the polygon exactly on the plane. (Though as noted earlier, to be completely correct we would have to store all polygons that are coplanar to the splitting plane in the current node.)

With the leafy BSP tree, no polygons are allowed to lie "on" the splitting plane. Instead, they must be assigned to either the front half space or the back half space. The choice is arbitrary, but must be consistent for all polygons. In other words, we "push" each polygon which lies on a splitting plane further down into the tree, so that it eventually ends up in a leaf node. The tricky part about pushing polygons down further into the tree is that pushing a polygon down into a node requires that we split the polygon against the receiving node's splitting plane. Splitting the polygon generally will result in two pieces, meaning that each smaller piece must then individually be pushed further down into the tree. In other words, with a leafy BSP tree, each child node's splitting plane is also allowed to split the polygons from parent nodes.

TEAM-FLY



NOTE The choice of whether polygons lying on the splitting planes are pushed to the front or the back half space influences whether the resulting convex polygon sets appear in the left or right child leaves of the tree.

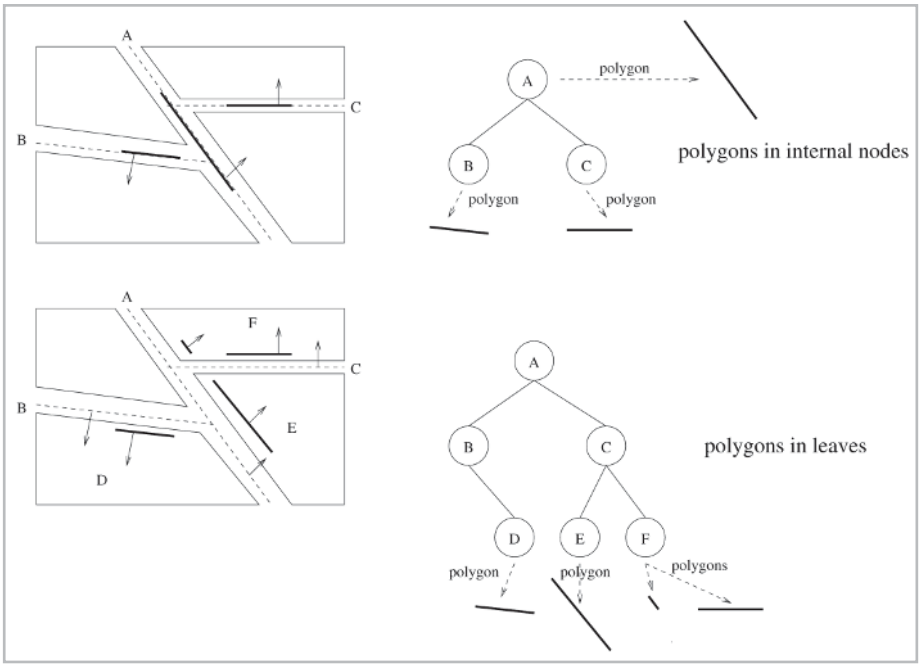


Figure 5-9: Pushing and splitting polygons down into the BSP tree until the various split parts finally appear only in leaf nodes. Polygons lying on the splitting planes are pushed into the front half space.

Team-Fly®

Rather than pushing each polygon individually through the tree, which would require multiple passes through the tree, we can save work by merging all split pieces coming from a higher-level tree node with the current polygon in the current node. In this way, as a polygon progresses down the tree, it gets clipped against each node's plane and merged with more and more polygons from each node along the way down. This causes clumps of polygons to form as they progress down the tree.

After pushing and splitting all polygons down into the leaf nodes of the tree, we end up with a BSP tree with polygons only at the leaves. In general, such a tree will have more nodes than the corresponding non-leafy version, because the polygons which were previously in the interior nodes have most likely been split into several pieces as they were pushed down the tree.

The interesting thing about a leafy BSP tree is that the polygons in each leaf node are all convex subsets of the original geometry. Recall that each node of a BSP tree represents a convex space because the slicing along infinite planes always preserves convexity of the space. Since all of the splitting planes were chosen from the planes of the polygons in the input set, all of the convex spaces have faces that align with the plane of at least one of the polygons of the input set. Furthermore, since all polygons lie on a splitting plane, then all post-slicing polygons lie on a face of their leaf node's convex space.

Convexity is an indication of geometric "simplicity," as we mentioned earlier. In particular, this means for any particular clump of polygons at the leaf of the leafy BSP tree, we can draw these polygons without sorting and in fact in any order, and be assured of a correct result. We only need to use back-face culling to eliminate drawing back-facing polygons. Since the polygons lie on the face of a convex space, none of the front-facing polygons can ever obscure each other.

The convexity of the regions is a fascinating geometric property, but there is still the question of visibility among the various convex regions. With the non-leafy BSP tree, this was solved because of the hierarchy of splitting planes and the recursive camera-based traversal order. We could use the same strategy with the leafy BSP tree, using the camera position to determine a traversal order of the leaf nodes, either from far to near (back to front) or near to far (front to back). However, the convexity of resulting polygon sets allows for other traversal possibilities. With a leafy BSP tree, solving the inter-region visibility problem can also be done by computing or specifying explicit connectivity information among the convex regions. This is the approach of portal-based schemes, which are covered later in this chapter.

Creating a Leafy BSP Tree

We can create a leafy BSP tree from a non-leafy BSP tree by using a recursive routine to push polygons down into their child nodes. The algorithm works as follows. Create a BSP tree as normal, with polygons stored in the internal nodes. Then, push each polygon down to a leaf node recursively as follows. Begin at the root of the entire tree with an empty convex clump.

1. Split the current convex clump against the current node's splitting plane. This yields two convex clumps: one in front of the splitting plane and one behind the splitting plane.
2. Assign the current node's polygon, which lies on the splitting plane, to one of the convex clumps. It doesn't matter which, but choose consistently. At this point, all polygons have been assigned either to the front or back convex clump.

3. Process the back half space as follows.
 3a. If the back convex clump is empty, no polygons currently lie within the back half space, but polygons deeper in the tree still will lie within the back half space. So, recursively call step 1 on the left child node, passing NULL as the convex clump.
 3b. If the back convex clump is non-empty, then this is the portion of the current convex clump that lies behind the splitting plane of the current node. We must continue to push this convex clump further down into the left child of the tree. There are two cases. If the left child node exists, then recursively call step 1 on the left child node, passing the back convex clump. This adds the back convex clump to the left sub-tree. If the left child node does not exist, then there previously was no polygon on the back side of the current splitting plane, but the current convex clump, which was pushed down from higher levels in the tree, now needs to exist within the left half space. So, create a new left child node, and assign the back clump to the new left child node. The newly created child node is then a leaf node.
4. Process the right half space in a similar fashion.
 4a. If the front convex clump is empty, then recursively call step 1 on the right child node, passing NULL as the convex clump.
 4b. If the front convex clump is non-empty, then check to see if the right child node exists. If the right child node exists, then recursively call step 1 on the right child node, passing the front convex clump. If the right child node does not exist, then create it, and assign the front convex clump to it. The newly created child node is a leaf node.

In this manner, each polygon from a higher-level node gets pushed down into a lower-level node. As the polygons get pushed down into the tree, they are always clipped against the current node's splitting plane, then (if they survive the clipping) they are merged with the polygon of the current node. Each polygon eventually gets assigned to a leaf node, though polygons coming from higher levels will likely be split into smaller pieces and scattered throughout the tree as they are pushed down.

Methods for Leafy BSP Trees in Class l3d_bsptree

The class `l3d_bsptree` contains methods for leafy BSP trees which we skipped earlier. Now, let's look at these methods again.

Method `push_polys_to_children` is a recursive routine that pushes polygons from the internal nodes down into the child nodes, using the algorithm outlined previously. Since the routine is recursive, the polygons all eventually end up (after possibly being split) in one or more leaf nodes.

Method `_permanently_merge_objects` merges two objects of type `l3d_object` into one. The reason we need this routine is to save work when pushing polygons down the tree; as mentioned earlier, if we join all split polygon fragments coming from a higher level in the tree with the polygon at the current level in the tree, we can form clumps of polygons which then all get pushed down simultaneously, rather than needing to make multiple passes through the tree with each single polygon. Forming clumps is done by joining the polygons in two separate `l3d_object` objects. The join operation takes two parameters: a destination object and a source

object. The destination object is the one which will contain the merged geometry afterwards; the source object is the object containing the additional polygons to be added to the current geometry in the destination object. Joining the objects requires us to create a new polygon in the destination for every polygon in the source, and to rewrite the new polygon's vertex index list to use the vertices in the destination's vertex list. It is possible that the copied polygon used vertices in the source object which do not yet exist in the destination object; in this case, we need to add the vertex to the destination's vertex list.

Method `make_videoscape_meshes` recursively traverses the leafy BSP tree and for each node with any polygons in it (for a leafy BSP tree this will be only the leaf nodes), it writes a separate Videoscape mesh file, with sequentially numbered filenames `vid0.obj`, `vid1.obj`, `vid2.obj`, and so forth. You can then load the meshes into Blender to see the individual convex meshes produced by leafy BSP tree. The next sample program illustrates this.

Sample Program: leafybsp

The sample program `leafybsp` reads in a Videoscape file as input, creates a leafy BSP tree based from the mesh, then writes out a series of Videoscape files, where each file contains the polygons in one leaf node of the leafy BSP tree. You can then load these files, all at once, into Blender, and see how the leafy BSP tree automatically divides arbitrary geometry into convex subsets.

The main program file is very similar to that of program `bsp`. The only difference is that after loading the Videoscape object file and creating the BSP tree, we call `push_polys_to_children` and `make_videoscape_meshes` to create and write the leafy BSP tree. Before and after the call to `push_polys_to_children` we call `print_stats` to show the statistics for the tree. Before pushing the polygons to the leaves, each node in the tree has exactly one polygon, as can be seen in the printed statistics. After pushing polygons to the leaves, the internal nodes have no polygons, and the leaf nodes have several polygons.

Listing 5-7: `main.cc`, main program file for program `leafybsp`

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../lib/geom/object/sector.h"
#include "../lib/geom/polygon/p3_flat.h"
#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rast3.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/geom/world/w_bsp.h"
#include "../lib/system/fact0_2.h"
#include "../lib/pipeline/pi_wor.h"
#include "../lib/dynamics/plugins/plugenv.h"
#include "../lib/geom/texture/tl_ppm.h"
#include "../lib/dynamics/plugins/vidmesh/vidmesh.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

```
#include <math.h>
#include <stdarg.h>
#include "../lib/geom/bsp/halfspace.h"
#include "../lib/geom/bsp/bsptree.h"

class my_world:public l3d_world_bsptree {
  protected:
    l3d_texture_loader *texloader;
    l3d_surface_cache *scache;
  public:
    my_world(void);
    virtual ~my_world(void);
};

my_world::my_world(void)
    : l3d_world_bsptree(320,240)
{
  l3d_screen_info *si = screen->sinfo;

  camera->VRP.set(0,0,-50,0);
  camera->near_z = float_to_l3d_real(1.0);
  camera->far_z = int_to_l3d_real(50);

  //- for mesa rasterizer's reverse projection
  rasterizer_3d_imp->fovx = &(camera->fovx);
  rasterizer_3d_imp->fovy = &(camera->fovy);
  rasterizer_3d_imp->screen_xsize = &(screen->xsize);
  rasterizer_3d_imp->screen_ysize = &(screen->ysize);

  int i,j,k;
  texloader = new l3d_texture_loader_ppm(si);
  scache = new l3d_surface_cache(si);

  //- create a plugin object
  l3d_object_clippable_boundable *oc;

  oc =
    new l3d_object_clippable_boundable(10);
  //- max 10 fixed vertices, can be overridden by plug-in if desired
  //- by redefining the vertex list

  oc->plugin_loader =
    factory_manager_v_0_2.plugin_loader_factory->create();
  oc->plugin_loader->load("../lib/dynamics/plugins/vidmesh/vidmesh.so");
  oc->plugin_constructor =
    (void (*)(l3d_object *, void *))
    oc->plugin_loader->find_symbol("constructor");
  oc->plugin_update =
    (void (*)(l3d_object *))
    oc->plugin_loader->find_symbol("update");
  oc->plugin_destructor =
    (void (*)(l3d_object *))
    oc->plugin_loader->find_symbol("destructor");
  oc->plugin_copy_data =
    (void (*)(const l3d_object *, l3d_object *))
    oc->plugin_loader->find_symbol("copy_data");

  texloader = new l3d_texture_loader_ppm(si);
  scache = new l3d_surface_cache(si);
```

```
  char plugin_parms[4096];
  sprintf(plugin_parms, "%d %d %d  1 0 0 0 1 0 0 0 1 soup.obj soup.ppm soup.uv",
          0,0,0);

  l3d_plugin_environment *e =
     new l3d_plugin_environment
       (texloader, screen->sinfo, scache, (void *)plugin_parms);

  if(oc->plugin_constructor) {
    (*oc->plugin_constructor) (oc,e);
  }

  oc->bounding_sphere.compute_around_object(oc);

  l3d_bsptree *tree;
  tree = new l3d_bsptree;
  tree->root = new l3d_halfspace;
  tree->root->bounding_sphere = oc->bounding_sphere;
  tree->root->object = NULL;
  tree->root->children.num_items = 0;
  tree->root->children[tree->root->children.next_index()] = 0;
  tree->root->children[tree->root->children.next_index()] = 0;

  tree->compute_for_object(tree->root, oc);
  tree->print_stats(tree->root);
  tree->push_polys_to_children(tree->root,NULL);
  tree->print_stats(tree->root);
  tree->make_videoscape_meshes(tree->root);

  delete tree;
   screen->refresh_palette();

  exit(0);
}

my_world::~my_world(void) {
  delete texloader;
  delete scache;
}

main() {
  l3d_dispatcher *d;
  l3d_pipeline_world *p;
  my_world *w;

  factory_manager_v_0_2.choose_factories();
  d = factory_manager_v_0_2.dispatcher_factory->create();

  w = new my_world();
  p  = new l3d_pipeline_world(w);
  d->pipeline = p;
  d->event_source = w->screen;

  d->start();

  delete d;
  delete p;
  delete w;
}
```

The program reads Videoscape file `soup.obj`. As output it writes several Videoscape files with filenames `vid0.obj`, `vid1.obj`, and so forth. To see what the program does, proceed as follows.

1. Load `soup.obj` into Blender. Notice that all polygons are in one mesh, and that the mesh is not convex (it has dents).
2. Run the sample program. It creates the files `vid0.obj`, `vid1.obj`, and so forth, in the current directory. (Before running the program you may wish to delete these files from the current directory.)
3. In Blender, erase everything by pressing **Ctrl+x**. Then, load file `vid0.obj`. This automatically also loads `vid1.obj`, `vid2.obj`, and so forth. Notice that the shape of the geometry is the same as in the original `soup.obj` file, but that the leafy BSP tree has automatically divided the geometry into convex subsets. In other words, instead of being one mesh, the geometry now consists of several convex meshes. Right-click individually on each mesh in Blender to see its convex shape.

In the binaries directory for the sample program `bsp`, there is another Videoscape mesh file, named `uglysoup.obj`. This mesh contains a number of random polygons with completely random 3D positions and orientations, often intersecting one another, and exemplifies the meaning of the words "polygon soup." Rename this file to `soup.ppm`, and rerun the program to see how the leafy BSP tree automatically creates convex regions even out of a completely random selection of polygons.

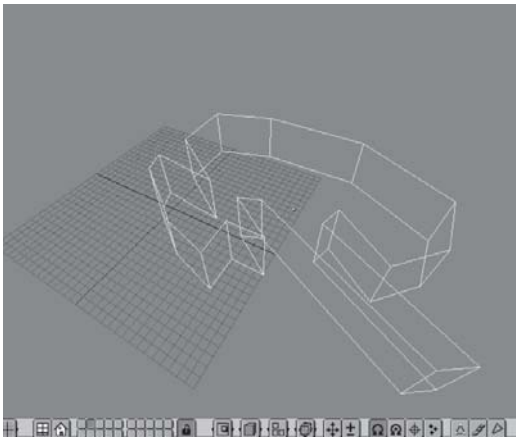


Figure 5-10: The original concave mesh in file `soup.obj`.

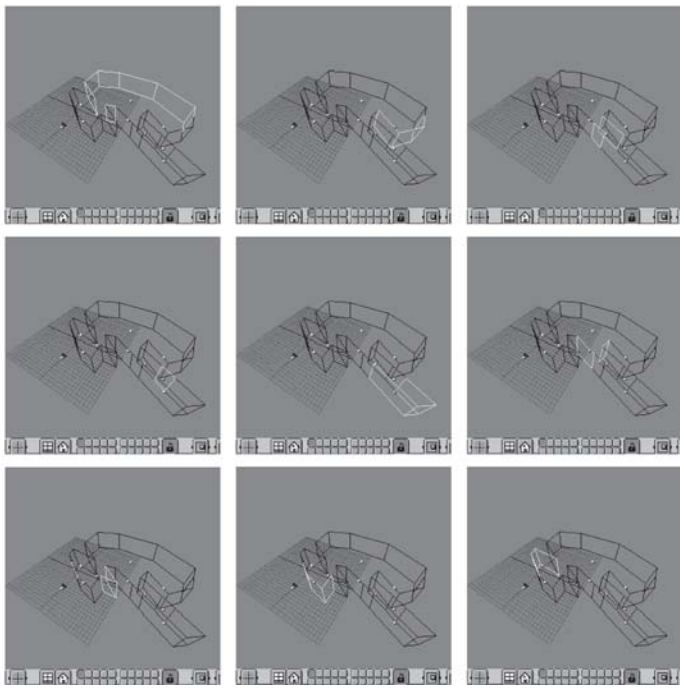


Figure 5-11: The automatic convex partitioning of the mesh by program `leafybsp`.

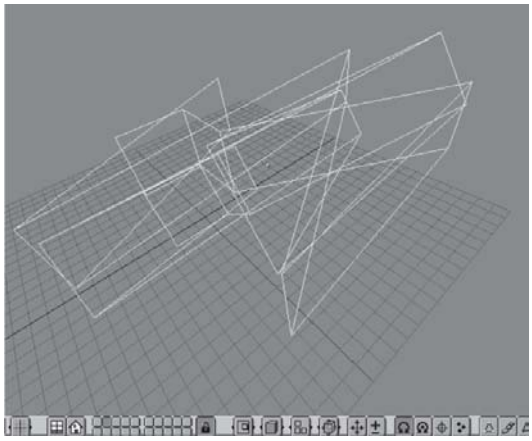


Figure 5-12: The chaotic concave, self-intersecting polygon soup in file `uglysoup.obj`.

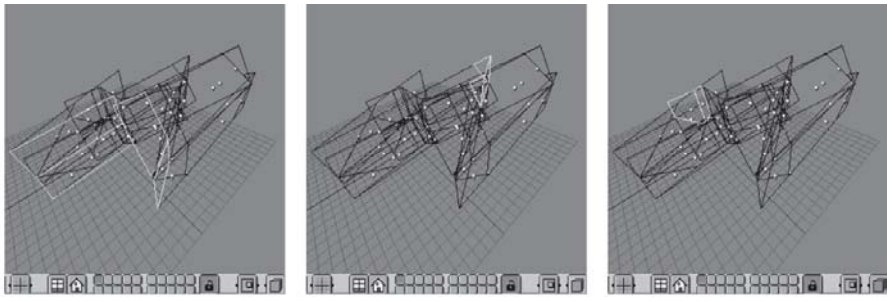


Figure 5-13: Some pieces of the automatic convex partitioning.



NOTE The program `leafybsp` usually creates a large number of convex regions, each with just a few polygons. Such a tree is not very efficient. By inspection, you can sometimes recognize better convex regions which might be able to be formed, but in general, forming better convex regions requires us to use some guiding heuristic when choosing the splitting plane, as discussed earlier.

Axis-aligned BSP Trees and Mini BSP Trees

The BSP tree structure we've considered so far divides space by splitting planes chosen from polygons within the input set. Another kind of BSP tree uses only axis-aligned planes to divide space [MOEL99]; in other words, each splitting plane must be parallel to one of the planes x=0, y=0, or z=0.

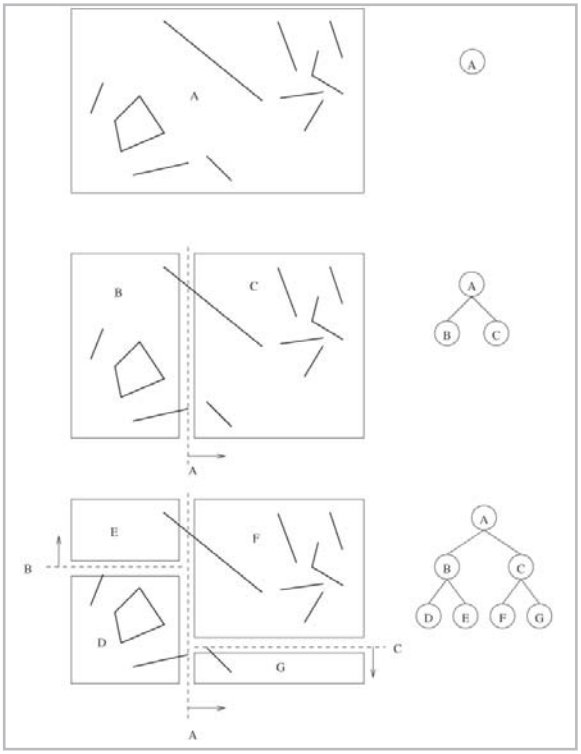


Figure 5-14: Axis-aligned BSP tree.

When a polygon crosses a splitting plane, we can split it into two pieces, just as we did before, so that each piece ends up in a separate half space. Alternatively, we can choose to view split polygons as lying "on" the splitting plane and store them at the splitting plane's level in the BSP tree, or we can assign the polygon to both half spaces [MOEL99].

The choice of the splitting plane is, given the additional restriction of axis-alignment, arbitrary. We can choose to split the space into exactly two equally sized halves, or try to balance the number of polygons in each half, or try to maximize the principle of locality, or try to minimize the

number of splits, or use any other heuristic that reduces the expected cost of desired operations on the BSP tree.

Previously, we chose the splitting planes from the polygons; when all polygons had been used, the tree was complete. With axis-aligned BSP trees, the choice of plane is completely arbitrary, so we need some other criteria to decide when to stop dividing space. For instance, we can stop when the depth of the tree reaches a certain maximum limit, or when the number of polygons in a space drops below a minimum limit, or when the physical size of a BSP tree node drops below a certain limit.

The axis-aligned BSP tree is more loosely connected with its underlying polygonal geometry than the BSP trees we looked at earlier. In particular, we cannot traverse the axis-aligned BSP tree to determine a completely correct back-to-front ordering for the polygons within the tree. Similarly, we cannot push polygons to the tree leaves and expect the formation of convex subsets.

What we can do with the axis-aligned BSP tree is use it for hierarchical view frustum culling. The volumes formed by the axis-aligned BSP tree are all axis-aligned boxes; we can compare the boxes themselves against the view frustum, as outlined previously, or we can compute a bounding sphere for each box and compare the sphere to the view frustum. We can also use the axis-aligned BSP tree to determine a back-to-front or front-to-back traversal of the nodes themselves (but not the polygons within them), using the camera position as we did before: first draw all nodes in the far half space as seen from the camera, then draw all nodes in the near half space. Each node, however, may contain one or more polygons, for which we have no ordering information. This means that for a correct display, we must resort to some other visibility mechanism within each node. For instance, we could draw all nodes back to front, and before drawing the polygons for each node, we could perform a z sort on the polygons within the node, as with the simple painter's algorithm. Another option is to draw all nodes front to back, and draw the polygons in each node in any order, using a z buffer to resolve visibility.

Another way of resolving the polygon ordering issue within one node is to compute a *mini BSP tree* for the node. A mini BSP tree is simply a small BSP tree for just the set of polygons contained within a particular node of the axis-aligned BSP tree. In general, a mini BSP tree is a polygon-aligned (not axis-aligned) BSP tree computed for some smaller subset of the polygons, where that subset as a whole is then subject to sorting or ordering at some higher level. In this case, we traverse the higher-level nodes of the axis-aligned BSP tree to determine a node ordering; then within each node, we use that node's mini BSP tree (which is not axis-aligned, but rather uses split planes of the polygons) to find a correct polygon ordering within that node. Taken as a whole, this yields a complete, correct visibility ordering.

BSP Tree as a Multi-resolution Solid-Modeling Representation

As we mentioned at the beginning of our discussion on BSP trees, we unfortunately cannot cover all the uses of BSP tree here; see the references in the Appendix for more information. But one last topic concerning BSP trees deserves mention: the use of the BSP tree as a representation of a polyhedral solid. Instead of storing a set of polygons defining the boundary of a polyhedral object, we can view the BSP tree as being a representation of the volume (and not the boundary) of the same

polyhedral object. In other words, a set of polygons and a BSP tree can represent the same geometry, in different ways; the polygons delineate the outer surface of the object, but the BSP tree can be interpreted as delineating the actual volume occupied by the object. This has a number of interesting implications, as we see shortly.

First, let's explore how a BSP tree represents volume. This requires us to extend the concept of a BSP tree, classifying nodes into what we will call *splitting nodes* and *non-splitting nodes*. Each node in a BSP tree stores a splitting plane (with the notable exception of the new nodes created at the leaves during generation of a leafy BSP tree—a topic we return to shortly). Thus, each node of a BSP tree splits space into a front and a back region. We call these nodes splitting nodes. The leaf splitting nodes (that is, those with no children) represent the finest (smallest) divisions of space within the BSP tree. For each leaf splitting node, let us add two children: a left child and a right child, each with no splitting plane. We call these nodes non-splitting nodes. These nodes represent the half spaces, which are not further divided, located behind and in front of the plane of the parent node, respectively. Remember that due to the slicing along infinite planes, all half spaces and thus all non-splitting nodes are convex.

After ensuring that every leaf splitting node has two non-splitting nodes as children, then all leaf nodes of the BSP tree are now non-splitting nodes. The union of the spaces represented by all non-splitting nodes is the entire space represented by the root of the tree; each non-splitting node is a small convex cutout of the entire space. Furthermore, each non-splitting node can be classified in terms of its position in relationship to its parent's splitting plane. Either the non-splitting node is a left child, behind its parent's plane, or it is a right child, in front of its parent's plane.

With a polygon-based BSP tree, we choose splitting planes from the planes of polygons within the input set. The interior of an object lies on one particular side of all polygons (either the front side or the back side); let us assume the polygon normals point outward, implying that the interior of the object lies on the back side of all polygons. Since all splitting planes come from the polygons, this means that the interior of the object always lies on the back side of all splitting planes. The back side of all splitting planes is the union of all non-splitting nodes which are left children of their parents. Therefore, the interior of the object (assuming outward-pointing normals) consists of all non-splitting nodes in the BSP tree which are left children. Similarly, the exterior of the object is the union of all non-splitting nodes which are right children.

In other words, a splitting node of a BSP tree divides space into two regions: a front and a back region. The region of space represented by a splitting node is inhomogeneous, being split further into a front and a back side. On the other hand, a non-splitting node represents a homogeneous region of space which is not further divided, and which belongs either entirely to the interior of the object (if it is a left child) or the exterior of the object (if it is a right child).

The leafy BSP tree which we saw earlier forces polygons from their parent nodes to be pushed down into their child nodes, being split as they move down the tree. This process will cause the creation of new tree nodes; a polygon coming down from higher in the tree will eventually get pushed into one of the half spaces of the leaf splitting nodes, thus creating a new non-splitting node as a child. Examine again the text output of program `leafybsp`; you will notice that before pushing polygons to the leaves, polygons exist both in the left and right nodes everywhere within the tree; after pushing, the polygons exist only in the right, leaf, non-splitting nodes of the tree.

Team-Fly®

(They exist in the right and not the left children because the polygon normals for the sample mesh point inward, and we push polygons on a plane to the back half space, not the front half space. Using a mesh with outward-pointing normals, and modifying class `l3d_bsptree` to push polygons on the plane to the front half space, causes the polygons to appear in the left non-splitting leaf nodes.)

Previously, we simply viewed the polygons within each leaf node of a leafy BSP tree as convex subsets of the original input polygons. We can now interpret each leaf node of the leafy BSP tree as a non-splitting node, thereby defining the convex volume of space resulting from the splitting of space along all planes in the nodes along the path to the non-splitting node. Instead of merely being a container for polygons, the leaf node itself is a convex space belonging to (or not belonging to, though we didn't create non-belonging nodes in the sample program) the volume of the object.

To summarize, we are interpreting the BSP tree as a hierarchy of bounding volumes instead of a hierarchy of splitting planes [NAYL98]. Indeed, we already encountered the idea of computing a bounding sphere for each node in the BSP tree. What we now can see is that each node in the tree is a bounding volume, and each non-splitting leaf node is a homogeneous volume which belongs either entirely inside or outside the object.

This hierarchy of bounding volumes means that checking for point containment within a particular region of space can be done quickly and hierarchically using the BSP tree. Essentially, the BSP tree can be used as a hierarchical spatial search structure.

Furthermore, we can *prune* the tree by eliminating some of the lower-level leaves and branches. Doing so creates a shallower tree with fewer nodes. If we interpret the nodes as a hierarchy of bounding volumes, pruning the tree means that we bound space less tightly and less exactly than we would if we went deeper in the tree. This means that the volume bounded by a pruned tree can be interpreted as a more "approximate" version of the geometry bounded by the full tree. Indeed, by pruning the tree, we can automatically create a model approximating the same geometry, but with fewer polygons [NAYL98]. This is useful for so-called *level of detail* techniques (see Chapter 9). Creating such reduced polygon models requires us to go from the BSP tree to a polygonal representation; previously, we went in the other direction, creating a tree from a polygonal model. To create a polygonal model from a (possibly pruned) BSP tree, we can start with a polygonal model of a bounding box, then slice it into pieces as we go down the tree. At each slicing stage, we have two half objects, and must also create two new polygonal faces, one for each half of the sliced object, to close the open holes left by the slicing operation. (Imagine slicing a milk carton in half, then gluing paper over the open areas.) We continue this slicing operation to an arbitrary depth in the tree, stopping whenever we want. When we stop, we then take all (convex) pieces either from all left children or all right children, depending on our surface normal orientation, and join them together. This gives us a polygonal representation of the volume bounded by the nodes in a BSP tree.

The idea of tree pruning to create a more approximate version of the bounded geometry again underscores the importance of using the principle of locality for creating good BSP trees. By emphasizing the principle of locality, a good BSP tree will, at higher levels, first immediately classify large regions of space as either belonging mostly inside or mostly outside the object under

question. At deeper levels of the tree, the spaces get divided further, and the inside/outside classification becomes more exact, until finally, at the homogeneous non-splitting leaf nodes, the classification is exact. A good BSP tree represents its underlying geometry in a multi-resolution fashion; a poor BSP tree indiscriminately splits and scatters detail throughout the tree.

BSP Trees and Dimension Independence

One last comment on BSP trees is that the concept can be applied to any dimensional space. The examples we looked at were 3D examples, where we used 2D splitting planes to divide 3D space. In general, for an n dimensional space, we use objects of dimension n–1 to divide the space into two pieces. The n–1 dimensional objects are referred to as *hyperplanes*. In 2D, for instance, we deal with 2D line segments and polygons, and split 2D space by 1D lines. (The game *Doom* used a 2D BSP tree.) In 1D, we deal with intervals, and split 1D space by 0D points. BSP trees can also be extended to dimensions greater than 3D, but 3D graphics, not 4D graphics, is the topic of this book.

Octrees

The octree is a structure very similar to the BSP tree, so we won't discuss it at great length here. The octree divides space hierarchically, just like a BSP tree, but each node has eight children, not two as with the BSP tree. The octree begins with a perfectly cubical bounding box around all of the geometry of interest. Then, the octree divides space into eight equally sized regions by slicing the cube exactly in half along three axis-aligned planes, in a manner very similar to the axis-aligned BSP tree. We can further recursively divide some or all child nodes in the same manner. As with the axis-aligned BSP tree, we must make decisions as to what to do when polygons cross splitting boundaries (split them, store them at that level, or store them in both children) and when to stop recursing (when the tree reaches a certain depth, when a node reaches a certain physical size, or when the polygon count for a node drops below a certain limit).

We can store the octree in memory by storing within each node pointers to all eight children. Alternatively, we can store the octree as a BSP tree, which enables us to use BSP tree algorithms more easily. To store the octree as a BSP tree, simply view each progressive octree subdivision of space as three consecutive BSP divisions of space; one along each of the planes parallel to x=0, y=0, and z=0. This storage scheme is a bit redundant, since the successive splits in a consecutive series of three splits all use the same planes in both the front and back half spaces, but it illustrates nicely the similarities between octrees and BSP trees.

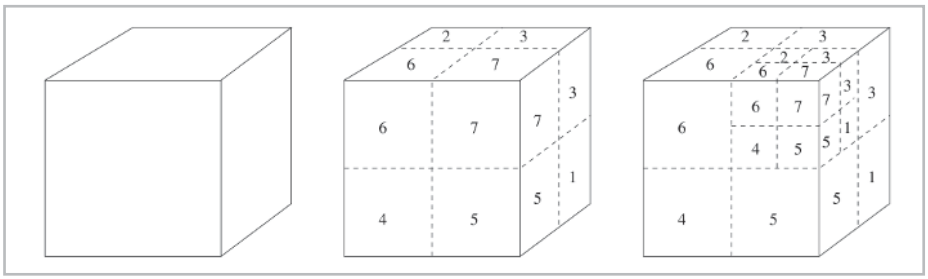


Figure 5-15: Octree division of space. We begin with an undivided cube (shown at the left of the diagram), then divide it into eight equally sized regions numbered 0-7 (shown in the middle part of the diagram; note that octant 0 is hidden), then again divide octant 7 into eight further octants (shown at the right of the diagram). The numbering scheme chosen for the octants is not standardized, and is taken from [FOLE92].

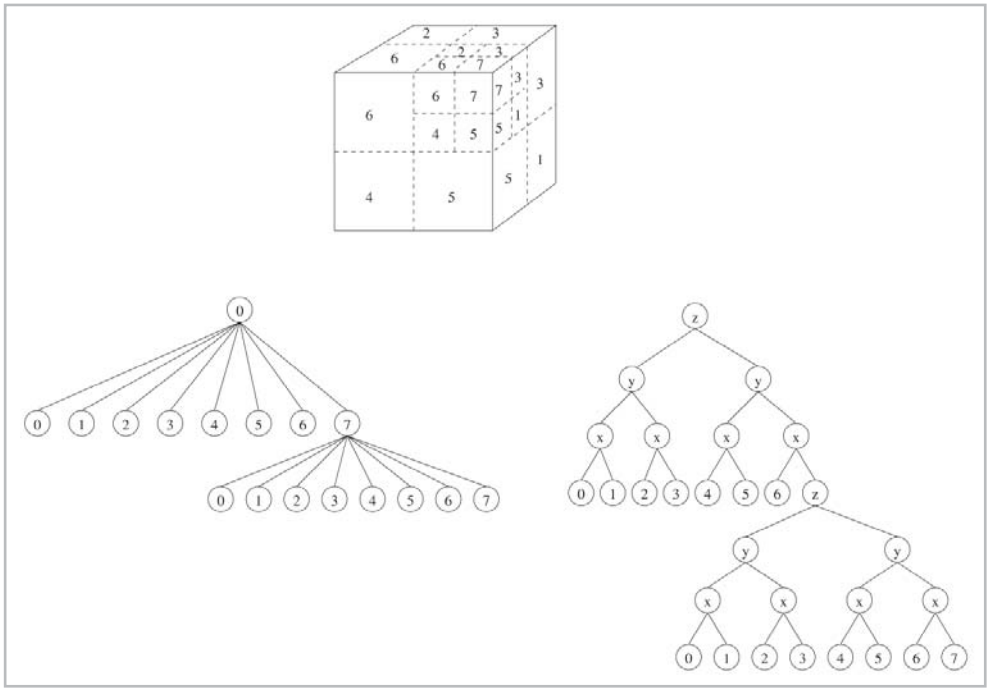


Figure 5-16: Storing an octree. We can store all eight children directly as child nodes of the parent node, or we can store the eight children within a BSP tree by splitting space three ways.

We can use the octree in a manner similar to the way we use an axis-aligned BSP tree. As with the axis-aligned BSP tree, there is no guarantee of convexity of the polygon groups within an octree node. So, to display the contents of an octree, we can traverse the nodes back to front or front to back, relying on a *z* sort, *z* buffer, or mini BSP tree to resolve the visibility question within

one node. We can also use the octree nodes hierarchically to achieve hierarchical view frustum culling.

Regular Spatial Partitioning

It's also worth mentioning that for some applications, a simple, non-hierarchical partitioning of space may suffice for VSD purposes. In this case, we simply take the bounding box of all geometry, and divide it equally into a fixed number of smaller boxes, say, 128 smaller boxes. We can store these linearly in an array. Essentially, this approach simply uses bounding volumes in a non-hierarchical manner. If the number of boxes grows to be too large, then some sort of hierarchy can be used to reduce the number of nodes that need to be traversed.

Even without introducing hierarchy into the data structure, we can use regular spatial partitioning in combination with a portal scheme so that only those boxes lying within the view frustum get traversed. This requires us to store links from each box to all of its neighbor boxes, and to traverse the boxes based on the viewpoint. Understanding this idea fully requires us to discuss portal schemes, which is our next topic.

Portals and Cells

Portal-based VSD schemes rely on an explicit specification of visibility to instantly cull invisible regions of space. They require a convex partitioning of space into cells, and the specification of connectivity information among cells via portals. Portals are typically useful for indoor environments, such as rooms, buildings, or tunnels; they are less useful for general, outdoor geometry such as landscapes, forests, or flight simulator scenery.

The idea behind portal-based visibility algorithms is a surprisingly simple and effective one. Let's start by looking at the three main components of the portal algorithm: the cell, the portal, and the current camera position.

The Main Ideas Behind the Portal Algorithm

A *cell*, sometimes called a *sector*, is a closed, convex set of polygons. It is easiest to think of a cell as representing the interior of a room, or some other enclosure. With portal schemes, the entire polygonal environment must be divided into separate convex cells (though the convexity restriction can be loosened, as we see later). The cell should be closed; there should be no "holes" in the cell where the absence of polygons would allow us to see through the cell to the outside. The reason for the closed nature of the cells is that visibility is strictly controlled with the portal algorithm—through portals.

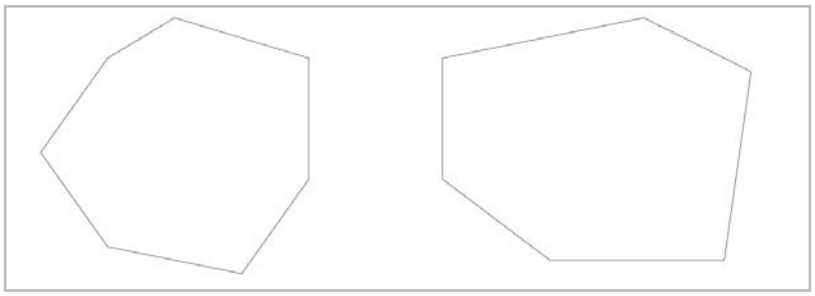


Figure 5-17: Two separate, convex cells.

A *portal* is a polygon that defines visibility and mobility between cells. It is easiest to think of a portal as a window cut out of the wall of a cell that allows us to see outside of the cell. By default, no cell can see any other cell. By introducing a *portal* polygon into a cell, we define a specific region of space which allows us to see and move into another specific cell. A portal, therefore, has a containing cell, a polygonal shape, and a target cell to which it leads. Notice that this definition is not necessarily two-way; a portal from cell A to cell B does not have to mean that a portal from cell B to cell A exists (though for a physically realistic world, this should be the case); in the real world, a window has the same shape, size, and connectivity regardless of which side we view it from. For fantasy worlds, we can allow one-way portals which allow us to see and move into rooms from which we can no longer return to the original room.

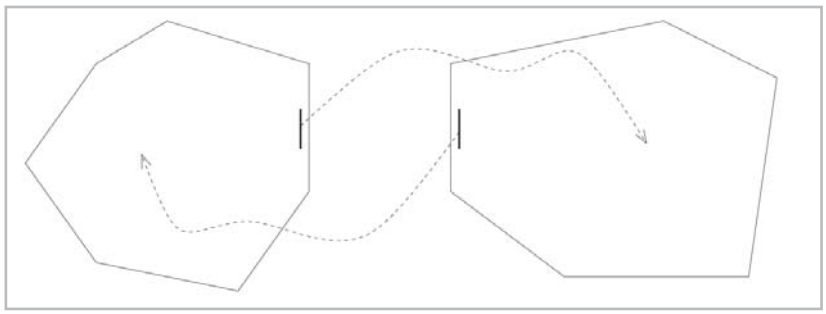


Figure 5-18: Adding a portal polygon to each cell, leading to the other cell.

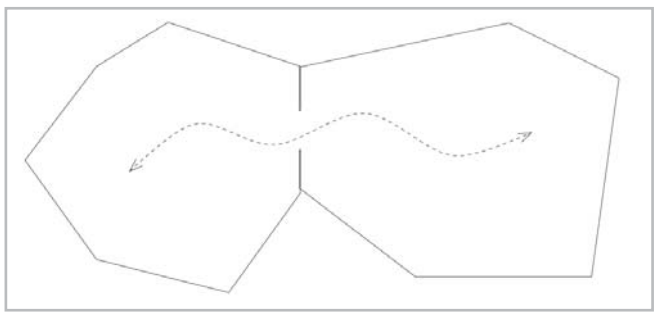


Figure 5-19: The effect of adding the portals is to make it appear that the cells are directly adjacent to one another, being connected by a "window" which is the portal polygon.

The current camera position is the third and final ingredient in the portal recipe. The camera is always located in exactly one cell within the world, and its position with respect to the cells and portals plays a vital role in the portal rendering algorithm.

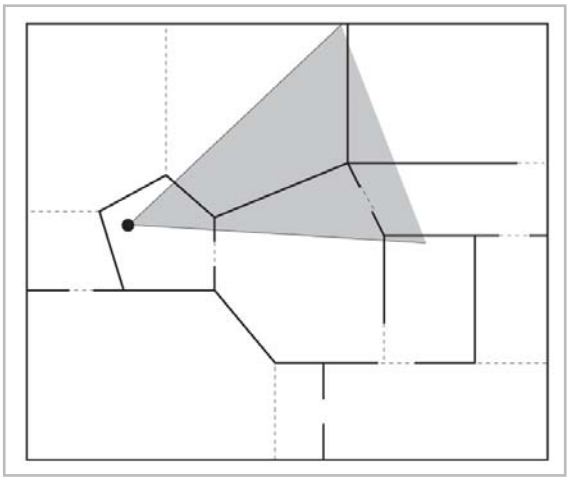


Figure 5-20: A portal model with several cells and portals. The camera is always located in exactly one cell. The gray area represents the view frustum, but note that not everything within the frustum is visible (see Figure 5-21). Notice that to preserve convexity, what appears to be one "room" in the diagram (delineated by heavy lines) is often composed of several convex cells, with "artificially" added portals (denoted by dashed lines).

Rendering a Portal World

Rendering a world with portals is simple. The idea relies on recursive rendering of and clipping to the screen space projection of portal polygons.

1. Set the viewing window to be a rectangle large enough to encompass the screen.
2. Render all polygons within the cell containing the camera, clipping them to the viewing window. Since the cell is convex, we only need to use back-face culling; all front-facing polygons can be drawn in any order, since they cannot obscure themselves.
3. While rendering, if we encounter a portal polygon, we set the view window to be the screen space projection of the polygon, and recursively call step 2 with the new view window and the cell on the other side of the portal.

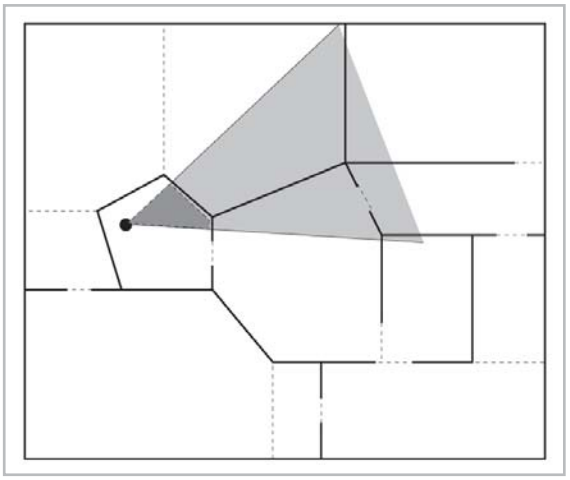


Figure 5-21: Rendering a portal scene traverses the cells based on the portals visible to the camera. The dark areas represent those portions of the view frustum visible to the camera. Notice that whenever a portal is encountered, what is behind the portal is also visible, through the portal (i.e., clipped to the portal). When a polygon is encountered, nothing behind it is visible (or even known to exist). For this viewpoint, exactly one cell is traversed.

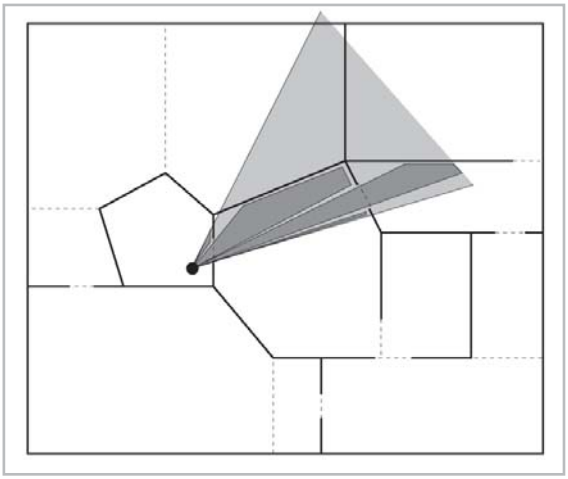


Figure 5-22: Another viewpoint in the same scene. Here, three cells are traversed. The light-shaded region represents the entire view frustum. The dark-shaded regions represent those portions of the world within the view frustum that are actually visible. Therefore, those cells that intersect the dark-shaded regions are the cells that are traversed. Notice that the dark-shaded regions (i.e., the zones of visibility) are blocked by solid polygons, but are allowed to pass through portal polygons.

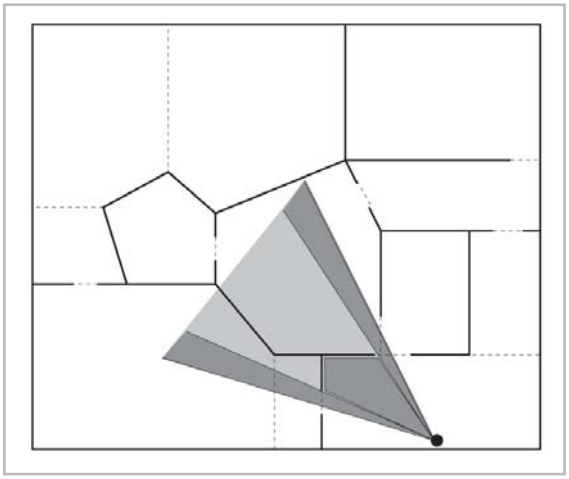


Figure 5-23: Another viewpoint. Here, five cells are traversed.

This means that if we see a portal polygon—in other words, if its projection lies within the current clip window—then we can see another part of the world. We thus recursively draw what is on the other side of the portal, using the portal's screen space projection as the new clip window. We must use the portal's outline as the new clip window because the portal limits what we can see; what is outside of the portal cannot be seen through the portal, for the current viewpoint. This is the key idea of the portal algorithm: we can only see other cells through the portal, meaning we clip to the portals. Anything not visible within the boundaries of a portal cannot be seen from the current viewpoint; a portal seen through a portal means a recursive visibility traversal.

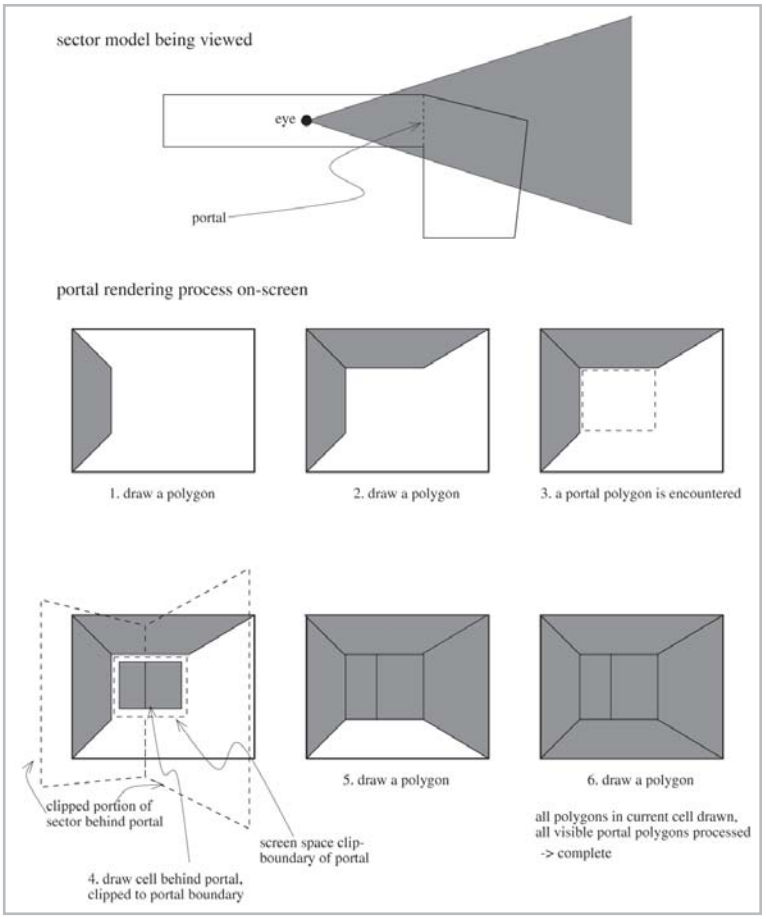


Figure 5-24: Portal rendering, as seen from the screen. Initially, the screen is empty. We draw polygons in any order; since they are convex, they cannot obscure one another. If we encounter a portal, we draw the contents of the connected cell, clipped to the portal boundaries.

Note that portal polygons, just like any other polygons, get clipped to the current clip window before being processed. This means that if we see a portal through a portal, the farther portal is, like all geometry seen through the portal, first clipped against the nearer portal. Recursively, this means that any portal that we see has already been clipped by all portals that came before it in the line of visibility. In this way, farther portals are always either smaller than the portal through which they are seen, or they are entirely clipped away by the nearer portal, in which case they are not processed further.

Notice how the portal scheme instantly culls any unseen parts of the world, which is essential for handling larger environments. If a portal is invisible, we do not spend any computational effort whatsoever examining any of the geometry on the other side of the portal. Recall that the world database in class `l3d_world` is by default a simple list of objects. In that case, we check each object, then discard it if it is invisible. Hierarchical view frustum culling can greatly reduce the number of such checks, but the idea is still the same: check, then reject; everything left over is visible. The portal scheme looks at the visibility situation in reverse. Initially, we start with the current cell and assume that everything outside the current cell is invisible. Only when we encounter a

portal do we then take the time to access and transform the data on the other side of the portal. Other culling schemes search the objects to see which ones are invisible; portal schemes assume everything is invisible, and search from a known starting point for portals and thus cells which are visible.

Put another way, the explicit division of the world into linked cells allows us to introduce the concept of a "current cell" from which we begin visibility computations. By always keeping track of the current cell, we always know where to begin our visibility computations; by following the links (portals), we find areas visible from the current cell. A good analogy can be made with the word processor. Imagine a word processor which does not store the current cursor location. Each time you type a character, the program would need to start at the first character in the document, then search all characters to see if they should be displayed on-screen or not. This would be intolerably slow. By organizing the document in terms of consecutive characters and lines, we then can keep track of the "current position" (the cursor location) and the currently visible region (all lines visible on-screen around the cursor). The idea is the same with portals; the cells and portals form a navigable graph of nodes where we can track the current position and use the current position as the starting point for further computation or navigation.

Observations about the Portal Scheme

Having understood the basics of portals, let's now look at some of the properties of this VSD approach.

Portals as a Connectivity Graph

Portals form an explicitly specified connectivity graph among various cells. We traverse the graph by determining which portals (in other words, which links among the graph nodes) are visible on-screen. Remember what we said earlier about the BSP tree: the typical data structures of computer science can be used to create effective visibility algorithms. Portals are another elegant example of this.

Advantages and Disadvantages

The main advantage of the portal scheme is that it allows us to quickly and efficiently cull large portions of the world in a very natural manner. We only draw what we see. If we don't see a portal, we do not draw it, and we do not waste time examining any part of the world database lying behind the portal. As soon as we look in the direction of the portal, we traverse that part of the world database and draw what is behind the portal. Also, if the cells are convex, there is no overdraw of pixels, which is important for performance in a software-based renderer. For the interior environments that portals are meant to handle, we also know that every pixel for each frame will be overwritten in the frame buffer. Since we view the cells from the inside, and since the cells are closed, the polygons and the portals for the current cell completely encompass the field of view and thus fill each pixel on screen (and exactly once). This means that we do not need to take the time to clear the frame buffer before drawing the next frame, which can provide a speedup for software rendering.

Another advantage is that we can choose to stop drawing once the recursion reaches a certain depth, providing a natural way of limiting scene complexity. The game *Descent* allowed you to control the depth of portal traversal via a configuration parameter (although it wasn't explicitly labeled as a traversal depth parameter). However, artificially limiting the traversal depth does mean that parts of the scene which would be visible simply do not get drawn, resulting in black "holes" in the picture. As the viewer approaches these regions (camera movement in portal environments is covered below), making them less distant and within the traversal threshold, they suddenly "pop" into view.

The main disadvantage of the portal scheme is the flip side of the same coin: the burden of explicitly finding or specifying connectivity information among convex regions of space. If we do not ensure convexity of the cells, then we must use some other VSD mechanism to draw the polygons within one cell correctly: a *z* buffer, sorting, or a mini BSP tree. But regardless of whether the cells are convex or not, we still must maintain connectivity information among the cells. We can do this by means of a modeling program designed for or adapted to portal modeling (see Chapter 8), or by using a BSP tree (discussed later in this chapter).

Back-Face Culling

When recursively traversing the portals, it is important to use back-face culling on the portal polygons also. This is because typically a portal leading from cell A to cell B has a counterpart portal in cell B leading to cell A. If we are in cell A looking through the portal at cell B, then we are facing the portal polygon, and the counterpart portal in cell B will be facing exactly away from the camera. Back-face culling will thus eliminate portals on the other side of the one we are looking at. If we do not eliminate back-facing portal polygons from consideration, then while drawing, we could traverse from A through a portal to B, through the counterpart portal back to A, then again through the first portal back to B, ad infinitum.

Clipping

Another item of note is the clipping to the portal boundaries. We can do this in 2D or in 3D. In 2D, we begin with a 2D clip window which is a rectangle the size of the screen. We project polygons, then clip them to the clip window. Portal polygons are projected and clipped to the current clip window, just like any other polygon. If we encounter a portal, we take the 2D projection of the portal polygon and use it as the new 2D clip window for rendering the cell on the other side. Any polygons (geometry or portal) that do not survive the clipping are discarded; those surviving are drawn (geometry) or traversed (portal).

In 3D, we clip polygons in world space against the planes formed by the edges of the portal polygon and the eye. As with the view frustum, we use the location of the eye and the two endpoints of the edge to form a plane, then clip against this plane. We repeat this for all edges of the portal polygon, in 3D world space. All polygons, including portal polygons, get clipped in 3D in this manner. A polygon surviving the clipping is then projected and drawn (for a geometry polygon) or is again used as a clipping polygon for the geometry behind it (for a portal polygon). In 3D, you can think of the portal clipping as clipping against a view frustum, but with no near and far plane, and with the same number of edges as the portal polygon.

Team-Fly®

Instead of performing exact clipping against the portal, we can also perform approximate clipping. To do this, we take the bounding box in screen space of the projected portal polygon, and use its four edges to form the top, bottom, left, and right planes of a frustum. This frustum is guaranteed to encompass more space than would the exact frustum of the portal polygon itself. We then clip against this approximate, larger frustum. This means that after clipping, edges of the polygons from the connected sector "stick out" or extend over the edges of the actual portal polygon, leading to an incorrect display. To solve this, we must use an additional VSD mechanism such as a *z* buffer. The reason we might want to perform approximate clipping is that 3D acceleration hardware sometimes offers acceleration for clipping against a normal view frustum (without the extra sides introduced by a portal frustum), as well as a hardware *z* buffer. Using accelerated frustum clipping and a hardware *z* buffer can be faster than analytically clipping the geometry to the exact portal edges or planes in software.

Convexity or Non-Convexity

Portals, as defined so far, deal with convex cells. The cells can also be non-convex, but this requires us to use another VSD algorithm in addition to the portal culling. In the case of non-convexity, the portal scheme can be viewed as a high-level rough ordering of the cells, as with the axis-aligned BSP tree or octree. Then within each cell, we resort to another VSD mechanism to draw the polygons within the cell correctly. In this case, the significant difference between portals and the axis-aligned BSP tree or octree is that cell-to-cell connectivity is explicit with portals.

Still, you might want to aim for convexity, depending on whether you plan to use software or hardware rendering. Part of the attraction of portals from a software rendering standpoint is that with exact clipping and convex cells, every pixel in the scene gets drawn exactly once, reducing overdraw. With non-convex cells, this advantage is lost.

Moving the Camera and Objects Within a Portal Environment

Of the VSD algorithms we've looked at so far, the portal scheme is the most explicit about dividing space into regions; the cells form an integral component of the approach. As mentioned earlier, the camera is always located in exactly one cell, and the entire portal rendering algorithm requires us to start in the cell containing the camera. This has two implications for a moving camera.

First, before rendering anything at all, we must determine which cell contains the initial camera position. We might know this information beforehand; for instance, if we always start the camera at position (0,0,0) in world space, and if we always create the first cell so that it encloses the point (0,0,0), then we know that the camera starts out in the first cell. Otherwise, we must search through the cells to see which one contains the camera. Since each cell is convex, we can do this by using a frustum-style point-to-plane check. The camera is within a cell if it is on the front side of the plane of every polygon defining the cell, as determined by evaluating each plane's equation with the camera position.

Second, we have to track the movement of the camera (or any other object which can move among cells) from cell to cell, so that we always know which cell the camera is in. There are two ways to do this. One is to constantly check, for every frame, which sector the camera is in, using

the plane tests as described above. This is not as time-consuming as it might sound; once we know the current camera position, it is a reasonable assumption that the camera can only move from the current cell to an adjacent cell. (The only way this could be otherwise is if the camera is moving so quickly that it takes a discrete step so large as to jump over an adjacent cell into one much farther away.) This means we don't need to search all cells to see which one the camera is in; we only need to search the cells adjacent to the current one—as well as searching the current cell—to determine the current camera location. Also, before checking for containment within each cell, we could use a simpler bounding volume (such as a sphere or cube) for each cell, and first check for containment within the bounding volume. If the camera is not within the cell's bounding volume, we don't need to expend the effort of doing the actual point-to-plane test for every polygon in the cell.

The other way of tracking camera movement is to track collisions with polygons. Whenever the camera moves through a polygon which is a portal, we set the current sector to be the one on the other side of the portal. The problem with this method is that checking to see if the camera crossed through a polygon is not completely trivial (see Chapter 8 for details). It does not suffice to check to see if the camera has moved from one side of the portal plane to the other side; there might be multiple portals within one plane, meaning that the camera might have moved through a different portal in the same plane. Tracking passage through portals requires us to intersect the 3D line segment, going from the previous camera position to the new camera position, with the polygon, and to see if the intersection lies within the polygon. We look at this intersection computation in detail in Chapter 8, when discussing collision detection.

Another movement issue with portals is how to handle moving objects. Typically, in a portal-based environment the geometry modeled by the cells remains static. We can associate moving objects with each cell by storing with each cell a list of objects contained in that cell. Drawing such objects can be easy or hard. It is easy if you use a *z* buffer; merely draw the objects for a sector as you encounter them. The *z* buffer sorts out all the depth issues. The issue is harder without a *z* buffer. The problem with drawing moving objects is that they destroy the convexity on which the portal algorithm relies. So, to draw such objects, we can make two passes for each cell—one to draw the convex sector geometry one to draw the objects on top of the cell geometry. Whenever we draw a cell, we first draw its geometry, and traverse any visible portals. Then, after the cell and all of its portals have been drawn, we draw the cell's moveable objects, which then appear in front of all other geometry and portals in the cell. Among the polygons of the object itself, we need to use some other means of VSD to ensure a correct display; the painter's algorithm is probably a good choice here.

As with the camera, if objects can move from cell to cell, we need to track their movement to notice when they move from one cell into another, so we can update the cell pointers which point to the contained objects for each cell.

Portals and the Near Z Plane

Clipping polygons against the near *z* plane is a necessity, as discussed in the introductory companion book *Linux 3D Graphics Programming*. However, it causes problems with portals. In particular, as we are approaching a portal, right as soon as we are about to cross it to move into the adjoining cell, the portal gets briefly but completely clipped away by the near *z* plane. This causes

a very distracting jump in the image just as we cross the portal. If we are clearing the frame buffer after every frame, then we will see a flash of black just before crossing the portal. If we do not clear the buffer after every frame (as noted before, it is not strictly necessary for indoor portal environments), then nothing gets drawn for a few frames while crossing the portal, leading to an apparent "freezing" of the image for a few frames.

The problem is that the camera's location is still within a cell, but just barely—it is right about

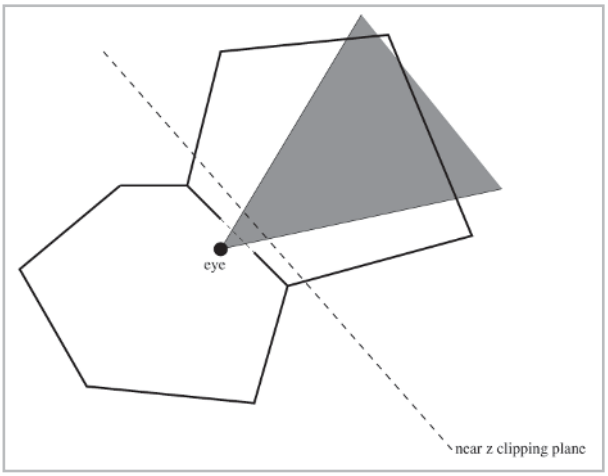


Figure 5-25: Here, a portal polygon is clipped away by the near z plane. Since the portal polygon is the only visible polygon in the current cell, and since the portal polygon is completely clipped away, in this case we see nothing on-screen.

to cross a portal into another cell, but it is not yet in the other cell. But we are so near to the portal polygon that it is clipped away by the near *z* plane; thus, the portal is not (or is only partially) drawn, meaning that we don't see what is behind the portal; indeed, we don't see anything at all.

To solve this problem, notice that when the camera is so close to a portal polygon so that the portal is clipped away, it is likely that the portal polygon is the only visible polygon from the cell containing the camera. This means that we can effectively consider the camera to already be in the adjoining cell. But we don't want to teleport the camera position to suddenly be inside of the cell; this would be just as visually jarring as the black flicker. Instead, we leave the camera in the same position, continuing to draw polygons based on this same camera position and orientation. What we change is the concept of the "current sector" for portal rendering. We do this by introducing a "pseudo-location" which is located slightly in front of the camera, along its VFW vector. In particular, the pseudo-location should be far enough in front of the camera so that when a portal polygon is near enough to be clipped away, the pseudo-location is already located on the far side of the near clipping plane. Then, for the purposes of portal traversal, we start in the cell containing the pseudo-location, not in the cell containing the actual location of the camera. In this way, the starting cell for the portal rendering algorithm is always the cell which is located a bit in front of the camera. Near z clipping and projection of polygons still take place with respect to the actual location of the camera; portal rendering, however, always starts on the far side of the near clip plane, meaning that we (almost) never have the situation that a single portal, providing visibility for the entire scene, is clipped away.

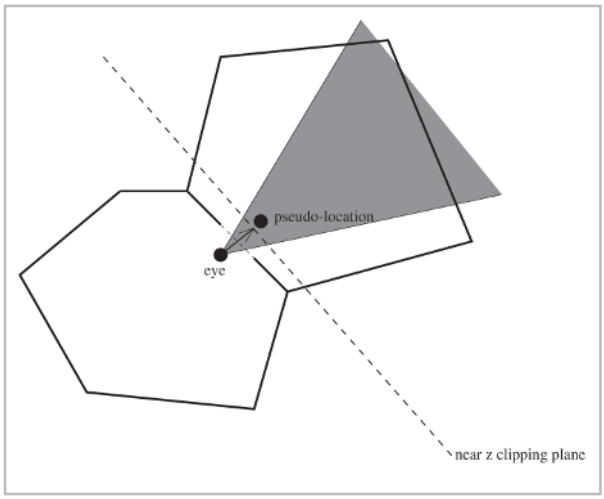


Figure 5-26: Introduce a pseudo-location slightly in front of the camera beyond the near z plane, and use its containing cell to start the portal rendering. This means the actual cell containing the camera is discarded, but this doesn't matter since we are almost outside the cell anyway.

Unfortunately, it is only "almost" never. It is still possible to view a portal polygon exactly or almost exactly on its edge; in such a case, both the camera location and the pseudo-location both lie on the plane of the portal polygon, and the portal still might be clipped against the near z plane, again resulting in a black screen for a moment. To solve this, we can do a number of things. First, we can simply ignore it. In many environments, it is not typical to stand within a portal polygon and look at it edge-on; it is much more common to pass through the portal more or less along its normal vector. In such a case, the pseudo-location approach provides an adequate solution. Another option is to detect when the viewer is looking at a portal edge-on, and to "bump" the camera slightly in front of or behind the portal. Finally, we could completely skip near z clipping for portal polygons. This only really causes a problem when the z value for one of the portal's vertices is exactly zero (because we cannot divide by z, necessary for the perspective projection, if z=0); if it is, then simply "jitter" it slightly to be, say, 0.0001. Of course, this results in slight inaccuracy of the drawing of the portal's border, but if you are that close to the portal, then it encompasses most of the screen anyway. The inaccuracy caused by a z jitter is likely less than a pixel and will likely last a maximum of one frame, so the inaccuracy is acceptable.

Shadows

Another interesting use of portals is the generation of shadows. With a portal-based world, the portals define the only visibility among cells. Anything not visible through a portal cannot be seen.

Computation of shadows can be viewed as a VSD problem. Imagine that the light source is like a camera. Then, all parts of polygons visible from the light source have their intensity affected by the light. Consequently, all other parts of polygons are not affected by the light, and are, by default, in shadow.

To use portals for shadow generation, we merely need to traverse the portal graph starting from the cell containing the position of the light. Since each cell is convex, no polygon in the cell can cause any part of the current cell to be in shadow; in other words, the convexity means that the cell cannot shadow itself. So, for all polygons within the current cell, we compute the light

intensity. With a light map based scheme, this means computing the light intensity for each lumel of each polygon's light map, as we saw in Chapter 2.

Then, for each portal polygon within the cell, we clip the polygons in the adjoining sector to the portal. It is easiest to do this clipping in 3D; for each edge of the portal polygon, we create a plane using the light location and the two endpoints of the edge. We then clip the adjoining sector to each such plane. Afterwards, we have exactly the parts of the polygons in the adjoining sector that are visible to the light source through the portal. We compute the light for these polygons as well, but only for the part of the polygon remaining after clipping to the portal. The process continues recursively. We can stop recursing either when no further portal polygons are visible, as we did when rendering the portals, or when the traversed sector lies beyond a certain physical distance from the light. Since we typically model light as having a maximum distance beyond which it has no effect, we can stop traversing portals if a sector is too far away to be affected by the light.

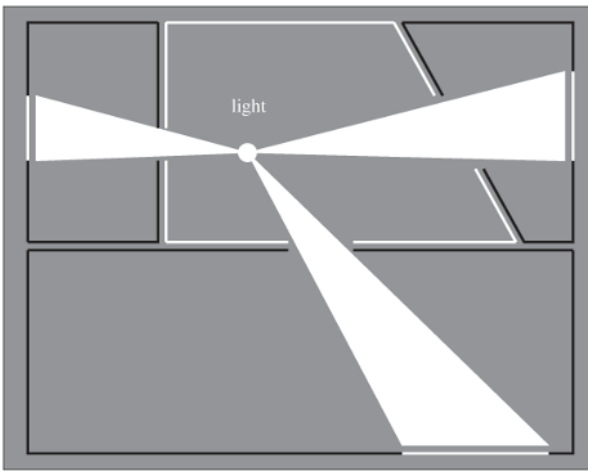


Figure 5-27: Using portals for shadow generation. All polygons in the current (convex) cell are lit. Then, from all adjacent cells, all parts of polygons visible through the portals are lit. This process continues recursively.

Let's briefly go over exactly how to compute the light intensities. We only compute light intensities for those parts of the polygons that survive after the clipping to a portal; only such post-clipping polygons are visible to the light. With a light map scheme, this means we need to transform the clipped polygon from 3D world space back into 3D light map space, using the world to texture space matrix of Chapter 2. Assuming that the 3D light map space lies exactly on the face of the polygon, which is how we have been defining it so far, then we can drop the third light map coordinate, and the 2D (u,v) light map coordinates of the clipped polygon are then the polygon's vertices in light map space. (If the light map space did not lie on the face of the polygon, we would need to do some sort of a projection, probably orthographic, to go from 3D light map coordinates (u,v,w) into 2D light map coordinates (u,v).) Since the light map coordinates are the polygon's vertices in the 2D light map, this means that we simply need to rasterize the polygon into the light map, using a rasterization routine similar to that developed in Chapter 2. The "frame buffer" for the rasterization is not the screen, but is rather the light map data itself. For each lumel lying within the polygon in light map space, we compute its intensity using any desired illumination formula from Chapter 2. Diffuse illumination is a good, simple choice to start with.

Remember that in Chapter 2 we mentioned that computing shadows with light maps can be time-consuming because we have to see if any polygon obstructs the light ray from the light source to the lumel under consideration. If any such polygon exists, which essentially requires us to search all polygons, then the lumel is in shadow; otherwise, it is lit. Portals reverse this computation by explicitly specifying lines of sight rather than trying to find blocked lines of sight. All lumels visible through a portal are in light; we leave all others alone, and they are by default in shadow.

Without light maps, there are other ways of using the portal-clipped polygons for rendering shadows. We can split the actual underlying geometry into two new pieces, the shadowed part and the lit part. The lit part would then be assigned a brighter texture; the shadowed part, a darker one. Or, we could split the geometry, store light intensities at the vertices of the split polygons—brighter intensities for the lit part, darker for the shadowed part—and use Gouraud shading. The split geometry needs to be re-joined and re-split whenever a light moves or changes, making this approach more suitable for static lights. With OpenGL, we can avoid splitting the underlying geometry by making two passes. In the first pass, we render the scene without lights, then in a second pass, we render the portal-clipped polygons (which are the parts of the polygons visible to the light) using OpenGL blending, exactly as we used a second OpenGL blending pass to render light maps. A second pass is only really feasible with hardware acceleration. With this scheme, you can think of the first pass as drawing the normal statically lit scene, and the second pass as painting extra light onto those parts of polygons which a dynamic light can see.

As this example illustrates, computing shadows can often be viewed as a VSD problem, only we must determine visibility from the light source, not the camera.

Mirrors

Portals can also be used to create mirror effects. A mirror can be considered to be a portal onto the same sector, merely seen from a different viewpoint. The image that a viewer sees in a (planar) mirror is the same image that the viewer would see if he were standing at the same position reflected against the plane of the mirror. This means that to create a mirror effect, we can have a portal that connects a sector to itself, but renders the sector from a new camera position—the reflected camera position. We could accomplish this by storing a flag with each portal to indicate if it is a mirror or not. If it is not, we render it as normal; if so, then we render it using the reflected camera position. Alternatively, we could store an arbitrary transformation matrix with each portal, allowing each portal to add an arbitrarily additional transformation to the geometry seen through the portal. This would allow not only mirrors, but also refractive effects.

Programming a portal-based mirror can be a bit tricky, since we must then render and transform different parts of the same sector multiple times (transformed once for the normal camera position, transformed again for the reflected camera position). We have to beware of infinite recursion if mirrors point at one another, creating a "hall of mirrors" effect. Also, the textures, light maps, and polygon orientations as seen from the reflected camera position all need to be reversed.

Portals and Other Rendering Methods

Portal rendering can be combined easily with other methods of rendering. If we allow each cell to be responsible for rendering itself, then when we encounter a portal, we ask the connected cell to render itself through a virtual function call. This is in contrast to an approach where one main controlling routine tries to render each sector manually by explicitly drawing polygons and traversing the portals.

Allowing each sector to render itself implies that each sector can use a different (even non-portal) algorithm for rendering. For instance, within a building, we could have portals for the doorways, leading from room to room. Each room would render itself in a normal portal-based manner, traversing the adjoining cells. Then, we could also have a doorway leading outside of the building, into an open landscape. This landscape could be stored as a cell or series of cells, but with an entirely different rendering algorithm. (Rendering landscapes is different than rendering indoor scenes, because the amount of data is greater and explicit partitioning into regions with exactly defined visibility is not generally possible.) Whenever we see the landscape regions through a portal, the portal asks the landscape region to render itself, thereby invoking (through the virtual function call) the appropriate landscape rendering routine. As we see later, the class `l3d_sector`, implementing a cell or a sector in the l3d library, declares its render method to be virtual, for exactly this reason.

Portals can also be used as a regular spatial partitioning method. To do this, we arbitrarily divide our scene geometry into a series of equally sized cubes. Each cube is then a cell. Then, we create a portal for the face of each cube, thus creating six portals (top, bottom, left, right, front, and back) for each cube. We connect each portal to the adjoining cube; the cubes on the very border of the scene will have no portals on their outermost faces. We then keep track of the current camera position, start rendering in the current cell, and recursively visit the neighboring cells. In general, the geometry contained within each cube will not be convex, meaning we have to resort to another VSD scheme to draw the contents of each cell correctly. Also, when rendering, we must artificially limit the portal traversal. In contrast to "normal" portals, which are small cut-out windows of visibility in otherwise opaque geometry, the portals in this case are always large enough to allow you to see to the end of the scene. The entire surface of every cell (in other words, the six faces of the cube) consists entirely of portals, meaning that anything clipped away by one portal appears in an adjacent portal, leading to essentially unlimited visibility. We can limit the depth of portal recursion, or set a maximum number of cells which will be rendered. Also, not rendering cells beyond the far z plane limits the number of visited cells.

The BSP tree can also be combined with portals. The leafy BSP tree, as we have seen, creates convex regions of space for an arbitrary input set. We can then find portals connecting these regions, and use portal traversal to render the scene. It should be noted, however, that using a BSP tree generally creates a tremendously large number of convex regions—many more than would result if you manually partitioned the world into convex regions (i.e., cells). Using a BSP tree is far from a completely automated, ideal solution to generating convex cells; for arbitrary geometry, generating a good convex partitioning automatically is a very difficult task. Nevertheless, let's briefly go over the BSP tree idea. After running polygons through the leafy BSP tree to obtain convex regions, we can find the portals connecting these regions by creating a large square polygon

for each splitting plane in the tree. The polygon must be large enough to encompass the geometry rooted at that node in the tree. Then, we push each square polygon down the tree, splitting it as we go down, just as we did when creating the leafy BSP tree, with the one change that a polygon lying on the split plane of a node gets pushed down both sides of the node, rather than an arbitrarily chosen side as we did earlier. After this process, in the leaves, we have polygon fragments which all are small convex cutouts of all of the splitting planes. The splitting planes split the original geometry into convex regions; this means that the portals, which must connect the convex regions, lie on the splitting planes. Indeed, they will be one of the splitting plane fragments we just created by slicing the large square polygons. All we need to do is search for a splitting plane fragment which appears in two leaf nodes; such a fragment is then a portal between the convex geometry of the two leaf nodes, because it is the part of the plane which split the geometry into those two convex parts.

Classes for Portals and Sectors

Let's now take a look at the l3d classes for portal rendering. The l3d classes use the term "sector" instead of "cell," so we'll continue to use this terminology in the following sections.

Class l3d_polygon_3d_portal

A portal is nothing more than a polygon which can be connected to a target sector. The class `l3d_polygon_3d_portal` portal therefore merely derives from the 3D polygon class `l3d_polygon_3d_clippable`, and adds a pointer to a sector, in member variable `connected_sector`. Member variable `connected_sector_name` stores a string identifier indicating with which sector the portal is connected; the reason for this is to facilitate loading portal worlds from a file, where a portal polygon might be created before the sector to which it is connected exists. In such a case, we first create all polygon, portal, and sector objects, storing the sector references as string. Then in a separate "linking" phase we actually assign values to the portal pointers to point to the appropriate sector objects, which are then searched for by their string identifier. Representing arbitrary references is always tricky for objects which must be streamed to and from an external source.

Listing 5-8: `p_portal.h`

```
#ifndef __P_PORTAL_H
#define __P_PORTAL_H
#include "../../tool_os/memman.h"

#include "poly3.h"
class l3d_sector;
class l3d_polygon_3d_portal :
      virtual public l3d_polygon_3d_clippable
{
  public:
    l3d_polygon_3d_portal(int num_pts):
    l3d_polygon_3d_clippable(num_pts),
    l3d_polygon_3d(num_pts),
    l3d_polygon_2d(num_pts)
  {}
    l3d_sector *connected_sector;
    char connected_sector_name[80];
```

```
    l3d_polygon_3d_portal(const l3d_polygon_3d_portal &r);
    l3d_polygon_2d *clone(void);
};

#include "../object/sector.h"

#endif
```

Listing 5-9: p_portal.cc

```
#include "p_portal.h"
#include "../../tool_os/memman.h"

l3d_polygon_2d* l3d_polygon_3d_portal::clone(void) {
  return new l3d_polygon_3d_portal(*this);
}

l3d_polygon_3d_portal::l3d_polygon_3d_portal(const l3d_polygon_3d_portal &r)
    : l3d_polygon_3d(r)
{

  connected_sector = r.connected_sector;

  strcpy(connected_sector_name, r.connected_sector_name);
}
```

Class l3d_sector

Class `l3d_sector` represents a region of space which can contain portals to other sectors. It is a logical unit of the virtual world; in other words, think of it as a "place," such as a room, or a hallway, or an airlock. It inherits from class `l3d_object_clippable_boundable`. In fact, the existing `l3d_object` class can already deal with portal polygons, since the object's polygon list contains pointers to abstract polygon objects, with which the new portal polygons are compatible. So, `l3d_sector` doesn't need to do anything to accommodate the portal polygons itself. Instead, `l3d_sector` adds methods and variables which are useful for portal rendering.

Member `objects` is a list of pointers to other objects of type `l3d_object`. This allows us to store moveable entities, such as spaceships, monsters, or rotating pyramids, within the sector. This reinforces the idea that a sector is a meaningful sub-unit of the virtual world, in which certain entities reside and act.

Since the sector can contain objects, the overridden `update` method calls `update` on each object, allowing each to update itself. The sector can also theoretically update its own geometry, meaning that a room might change shape over time, for instance.

The overridden assignment operator copies all sector data from the source to the target, so that the target is an identical but separate copy of the sector.

The new virtual method `render` implements the portal rendering scheme. This is a recursive method that takes a 2D clip window as a parameter. For all polygons in the sector, we transform them into world and camera space, apply back-face culling, then perform a perspective projection, just as we have been doing so far. Then, we clip the polygon to the current clip window. Initially, the clip window is a rectangle as large as the screen (as we saw in Chapter 2); but with portal rendering, this clip window becomes smaller and smaller as we render sectors which are further and further away. Before drawing a polygon, we check to see if the polygon is a portal polygon. If it is,

we recursively call `render` on the portal's connected sector, passing the portal polygon itself, whose geometry has now been projected into screen space, as the clipping window. This means that the screen space projection of the portal polygon limits the boundaries of what can be seen through the portal, which is the whole idea of portal rendering.

The only tricky part about the method `render` is the combination of the objects' polygon lists and the sector's polygon list. As we mentioned before, we should draw moveable objects in a cell after drawing that sector's geometry, so that the objects appear on top of the sector's geometry. We do this by joining all the polygon nodes of all objects together into one big list, then appending this object polygon node list to the polygon node list of the sector. In this way, we first process and draw the sector polygons, then draw and process the object polygons. Remember that for a correct display of object polygons we must use some other VSD mechanism to sort the polygons within and among the objects, but here we simply draw them in any order. When joining the object and sector polygon lists, we have to set the transformation stage variable (which tracks how many times an object, polygon, or vertex has been transformed) for the object polygons to be the same as that of the sector polygons. In general, an object polygon may have been transformed more times than the sector polygon, because the object may undergo its own local transformation (it may spin itself or move around, for instance). But when we join the object polygon list with the sector polygon list, all polygons should have the same transformation stage for consistency. The code in method `render` therefore takes care of joining the object and sector polygon lists, and ensuring that their transformation stage counters are the same.

Note that the method `render` is virtual; as mentioned earlier, this means that descendants of the sector class can implement new rendering methods for different types of sectors.

Listing 5-10: `sector.h`

```
#ifndef __SECTOR_H
#define __SECTOR_H
#include "../../tool_os/memman.h"

#include "obound.h"
#include "../../view/camera.h"

class l3d_polygon_3d_portal_clippable;

class l3d_sector :
      public l3d_object_clippable_boundable
{
  public:
    l3d_sector(int num_fixed_vertices) :
    l3d_object(num_fixed_vertices),
    l3d_object_clippable(num_fixed_vertices),
    l3d_object_clippable_boundable(num_fixed_vertices)
    {
      objects = new l3d_list<l3d_object_clippable_boundable *>(1);
    };
    virtual ~l3d_sector(void);
     l3d_list<l3d_object_clippable_boundable *> *objects;
    int update(void);

    virtual void render(l3d_polygon_2d *clipwin,
                        l3d_rasterizer_3d *rasterizer, l3d_camera *camera,
                        int xsize, int ysize);
```

Team-Fly®

```
    l3d_sector& operator= (const l3d_sector &r);

};

#include "../polygon/p3_clip.h"
#include "../polygon/p_cport.h"

#endif
```

Listing 5-11: sector.cc

```
#include "sector.h"
#include "../../tool_os/memman.h"

l3d_sector & l3d_sector::operator= (const l3d_sector &r) {
  strcpy(name, r.name);
  parent = r.parent;

  num_xforms = r.num_xforms;
  modeling_xform = r.modeling_xform;
  for(int i=0; i<r.num_xforms; i++) {
    modeling_xforms[i] = r.modeling_xforms[i];
  }

  *vertices = *r.vertices;

  polygons = r.polygons;
  for(int i=0; i<r.polygons.num_items; i++) {
    l3d_polygon_3d *p = dynamic_cast<l3d_polygon_3d *>(r.polygons[i]->clone());
    if(p) {
      p->vlist = &vertices;
      polygons[i] = p;
    }else {
      printf("error during cloning: somehow a non-3d polygon is in an object!");
    }
  }

  polygon_node_pool = r.polygon_node_pool;

  nonculled_polygon_nodes = NULL;
  for(int i=0; i<r.polygon_node_pool.num_items; i++) {
    if(r.nonculled_polygon_nodes == &r.polygon_node_pool[i]) {
      nonculled_polygon_nodes = &polygon_node_pool[i];
    }
  }

  transform_stage = r.transform_stage;
  strcpy(plugin_name, r.plugin_name);
  plugin_loader = r.plugin_loader;
  plugin_constructor = r.plugin_constructor;
  plugin_update = r.plugin_update;
  plugin_destructor = r.plugin_destructor;
  plugin_data = r.plugin_data;

  objects = r.objects;
   return *this;
}

l3d_sector::~l3d_sector(void) {
  delete objects;
```

```
}

void l3d_sector::render(l3d_polygon_2d *clipwin,
                        l3d_rasterizer_3d *rasterizer, l3d_camera *camera,
                        int xsize, int ysize)
{
  static int level=0;
  level++;

  int iObj, iFacet;

  l3d_vector facet_to_cam_vec;
  l3d_vector N;

  l3d_polygon_3d_clippable *pc;

  //- init polys for all objects contained in the sector

  for(int i=0; i<objects->num_items; i++) {
    (*objects)[i]->init_nonculled_list();
    (*objects)[i]->vertices->num_varying_items = 0;
    (*objects)[i]->transform_stage = 0;

    for(register int ivtx=0; ivtx<(*objects)[i]->vertices->num_fixed_items; ivtx++) {
      (*((*objects)[i]->vertices))[ivtx].reset();
    }

    l3d_polygon_3d_node *n;
    n = (*objects)[i]->nonculled_polygon_nodes;
    while(n) {
      n->polygon->init_clip_ivertices();
      n->polygon->init_transformed();
      n=n->next;
    }

    //- position all vertices of object into world space

    if ((*objects)[i]->num_xforms) {
      (*objects)[i]->transform((*objects)[i]->modeling_xform);
    }

    //- sector transform
    if (num_xforms) {
      (*objects)[i]->transform(modeling_xform);
    }
  }

  //- now concatenate all polys in all objects into one big object-polys
  //- list.

  l3d_polygon_3d_node *n_objects=0;

  for(int i=0; i<objects->num_items; i++) {

    //- prepend this object's polys to the overall list of all polys in all
    //- objects
    if((*objects)[i]->nonculled_polygon_nodes) {
      l3d_polygon_3d_node *n_temp;
      n_temp = (*objects)[i]->nonculled_polygon_nodes;
      while(n_temp->next) {
```

```
      n_temp=n_temp->next;
    }

    n_temp->next = n_objects;
    if(n_objects) {n_objects->prev = n_temp; }
    n_objects = (*objects)[i]->nonculled_polygon_nodes;

    n_objects->prev = NULL;
  }
}

//- initialize polys for the sector geometry
init_nonculled_list();
vertices->num_varying_items = 0;
transform_stage = 0;

for(register int ivtx=0; ivtx<vertices->num_fixed_items; ivtx++) {
  (*(vertices))[ivtx].reset();
}

l3d_polygon_3d_node *n;
n = nonculled_polygon_nodes;
while(n) {
  n->polygon->init_clip_ivertices();
  n->polygon->init_transformed();
  n=n->next;
}

//- position all vertices of sector into world space

if (num_xforms) {
  transform(modeling_xform);
}

//- at this point, the sector geometry and the geometry of all
//- contained objects are in world coordinates and have any transformations
//- already applied. now we dump all polys from all objects and the sector
//- into one big list - from now on, there is no distinction between
//- sector polys and contained-object-polys. they are just polys from now
//- on. note that the object polys come after the sector polys in the list.
{
  l3d_polygon_3d_node *n;

  //- for all object polygons and their vlists: set the transform stage
  //- to be the same as for the sector.

  n = n_objects;
  while(n) {
    n->polygon->set_transform_stage(transform_stage);
    n = n->next;
  }

  n = nonculled_polygon_nodes;
  if(n) {
    while(n->next) {n = n->next; }
    n->next = n_objects; //- append object polys to sector poly list
    if(n_objects) {n_objects->prev = n; }
  }else {
    nonculled_polygon_nodes = n_objects;
  }
```

```
  }

  camera_transform(camera->viewing_xform);

  n = nonculled_polygon_nodes;

  while(n) {

    N.set(n->polygon->sfcnormal.transformed.X_ - n->polygon->center.transformed.X_,
          n->polygon->sfcnormal.transformed.Y_ - n->polygon->center.transformed.Y_,
          n->polygon->sfcnormal.transformed.Z_ - n->polygon->center.transformed.Z_,
          int_to_l3d_real(0));

    facet_to_cam_vec.set (0 - n->polygon->center.transformed.X_,
                          0 - n->polygon->center.transformed.Y_,
                          0 - n->polygon->center.transformed.Z_,
                          int_to_l3d_real(0));

    /* dot product explicitly formulated so no fixed point overflow */
    /* (facet_to_cam_vec is very large) */
    /* (use of the inline function "dot" uses default fixed precision) */
    /* equivalent to: if(N.dot(facet_to_cam_vec) > 0 ) */
    if(

#define BACKFACE_EPS float_to_l3d_real(0.001)
      (dot(N,facet_to_cam_vec) > BACKFACE_EPS )

      &&

      n->polygon->clip_near_z(camera->near_z)

    )

    {
    }else {

      if(n->prev) {
        n->prev->next = n->next;
      }else {
        nonculled_polygon_nodes = n->next;
      }

      if(n->next) {
        n->next->prev = n->prev;
      }
    }

    n = n->next;
  }

  apply_perspective_projection
  (*camera, xsize, ysize);

  int ncount=0;
  n = nonculled_polygon_nodes;
  while(n) {

    if(n->polygon->clip_to_polygon_2d(clipwin))
    {
```

```
      l3d_polygon_3d_portal *p;
      if ( (p=dynamic_cast<l3d_polygon_3d_portal *>(n->polygon)) ) {

        p->connected_sector->render(p,rasterizer,camera,xsize,ysize);
      }else {
        ncount++;
        n->polygon->draw(rasterizer);
      }
    }
    n = n->next;
  }

  level--;
}

int l3d_sector::update(void) {
  for(register int i=0; i<objects->num_items; i++) {
    (*objects)[i]->update();
  }
  return 1;
}
```

Class l3d_world_portal_textured_lightmapped_obj

Class `l3d_world_portal_textured_lightmapped_obj` is a new world class which handles portal rendering with support for texturing, light mapping, and plug-in objects. It inherits directly from class `l3d_world`. Originally, this class descended indirectly from three simpler classes, `l3d_world_portal` (portal rendering without texture mapping), `l3d_world_portal_textured` (portal rendering with textured polygons), and `l3d_world_portal_textured_lightmapped` (portal rendering with textured and light mapped polygons). While it is a useful exercise to add these features step by step, all of these classes are actually very similar; the main difference is merely the types of polygons they create and the way they read them in from a file. It would take too much time and space here to look at all of the simpler classes, without really serving any educational purpose, so I decided to make the class `l3d_world_portal_textured_lightmapped_obj` independent of the simpler classes, so that it could be seen and understood by itself. On the CD-ROM, you can still find the simpler portal classes, and sample programs using them (the simpler sample portal programs are named `portal`, `portex`, and `porltex`, for portal, textured portal, and textured and light mapped portal rendering).

Listing 5-12: `w_porlotex.h`

```
#ifndef __W_PORLOTEX_H
#define __W_PORLOTEX_H
#include "../../tool_os/memman.h"

#include "world.h"
#include "../object/sector.h"
#include "../texture/tl_ppm.h"
#include "../surface/scache.h"
#include "../polygon/p_cport.h"

class l3d_world_portal_textured_lightmapped_obj :
      public l3d_world,
      public l3d_texture_computer
```

```
{
  protected:
    l3d_texture_loader *texture_loader;
    l3d_list<l3d_texture_data> tex_data;
    l3d_surface_cache *scache;

  public:
    l3d_sector *sector_by_name(const char *name);
    l3d_sector *current_sector;

    l3d_world_portal_textured_lightmapped_obj(int xsize, int ysize);
    ~l3d_world_portal_textured_lightmapped_obj(void);
    virtual void load_from_file(const char *fname);
    void place_lamp(l3d_sector *sector, l3d_polygon_3d_clippable *clipwin);
    void reset_surface_cache(void) {
      scache->reset();
    }

    void update_all(void);
    void draw_all(void);
};

#endif
```

Listing 5-13: w_porlotex.cc

```
#include "w_porlotex.h"
#include "../object/object3d.h"
#include "../polygon/p_portal.h"
#include <string.h>
#include "../polygon/p3_ltex.h"
#include "../polygon/p3_cflat.h"
#include "../polygon/p_cport.h"
#include "../../raster/ras_swl.h"
#include "../../view/camcol.h"
#include "../../dynamics/plugins/plugenv.h"
#include "../../system/fact0_2.h"
#include "../../tool_os/memman.h"

void l3d_world_portal_textured_lightmapped_obj::update_all(void) {
  l3d_world::update_all();

  int i;
  l3d_sector *s;
  int found=0;

  int portal_num=0;
  s = current_sector;
  while(!found) {

    s->reset();

    if (s->num_xforms) {
      s->transform(s->modeling_xform);
    }
     s->camera_transform(camera->viewing_xform);

    found=1;
    for(i=0; i<s->polygons.num_items && found; i++) {
      l3d_polygon_3d *p;
      l3d_vector N, facet_to_cam_vec;
```

```
      p = s->polygons[i];
      N.set(p->sfcnormal.transformed.X_ - p->center.transformed.X_,
            p->sfcnormal.transformed.Y_ - p->center.transformed.Y_,
            p->sfcnormal.transformed.Z_ - p->center.transformed.Z_,
            int_to_l3d_real(0));

      facet_to_cam_vec.set (0 - p->center.transformed.X_,
                            0 - p->center.transformed.Y_,
                            0.5 - p->center.transformed.Z_,
                            int_to_l3d_real(0));

      /* dot product explicitly formulated so no fixed point overflow */
      /* (facet_to_cam_vec is very large) */
      /* (use of the inline function "dot" uses default fixed precision) */
      l3d_real d;
      d=dot(N,facet_to_cam_vec);
      if(d>-0.05)

      {
      }else {
        found = 0;
      }
    }
    if (!found) {

      if(s!=current_sector) portal_num++;

      int next_sector_found=0;
      l3d_polygon_3d_portal *por;
      while(portal_num<current_sector->polygons.num_items && !next_sector_found) {
        if ( (por = dynamic_cast<l3d_polygon_3d_portal *>
                    (current_sector->polygons[portal_num])) )
        {
          next_sector_found = 1;
          s = por->connected_sector;
        }else {
          portal_num++;
        }
      }

      if(!next_sector_found) {
        found = 1;
      }
    }else {
      current_sector = s;
    }
  }
}

void l3d_world_portal_textured_lightmapped_obj::draw_all(void) {
  current_sector->render(screen->view_win,
                         rasterizer, camera,
                         screen->xsize, screen->ysize);

}

l3d_world_portal_textured_lightmapped_obj::
l3d_world_portal_textured_lightmapped_obj
(int xsize, int ysize)
    : l3d_world(xsize, ysize)
```

```
{

  l3d_screen_info *si = screen->sinfo;
  si->ext_to_native(0,0,0); //- allocate black bg color first
  texture_loader = new l3d_texture_loader_ppm(screen->sinfo);

  rasterizer_3d_imp->fovx = &(camera->fovx);
  rasterizer_3d_imp->fovy = &(camera->fovy);
  rasterizer_3d_imp->screen_xsize = &(screen->xsize);
  rasterizer_3d_imp->screen_ysize = &(screen->ysize);

  scache = new l3d_surface_cache(screen->sinfo);

}

l3d_world_portal_textured_lightmapped_obj::
~l3d_world_portal_textured_lightmapped_obj(void)
{
  delete scache;
  delete texture_loader;
}

l3d_sector *l3d_world_portal_textured_lightmapped_obj::
sector_by_name(const char *name)
{
  int i;
  int found=0;
  l3d_sector *result = NULL;

  for (i=0; (i<objects.num_items) && (!found); i++) {
    if ( strcmp(objects[i]->name, name) == 0 ) {
      result = dynamic_cast<l3d_sector *> ( objects[i] );
      found=1;
    }
  }

  if(result==NULL) printf("NULL sector %s!",name);
  return result;

}

void l3d_world_portal_textured_lightmapped_obj::load_from_file(const char *fname) {
  FILE *fp;

  int linesize=1536;
  char line[1536];
  int tex_count, snum;

  fp = fopen("world.dat", "rb");

  fgets(line,sizeof(line),fp);
  sscanf(line,"%d", &tex_count);

  for(int i=0; i<tex_count; i++) {
    fgets(line,sizeof(line),fp);
    while(line[strlen(line)-1] == '') {line[strlen(line)-1] = '0'; }

    texture_loader->load(line);
    int next_tex_index;
    next_tex_index = tex_data.next_index();
```

```
  tex_data[next_tex_index].width = texture_loader->width;
  tex_data[next_tex_index].height = texture_loader->height;
  tex_data[next_tex_index].data = texture_loader->data;
}

screen->refresh_palette();

fgets(line,linesize,fp);
sscanf(line,"%d",&snum);

l3d_sector *current_sector=NULL;

for(int onum=0; onum<snum; onum++) {
  char sname[80];
  char keyword[80];
  char plugin_name[1024];
  char rest_parms[4096];
  int numv, nump;
  fgets(line, linesize, fp);
  strcpy(keyword,"");
  strcpy(sname,"");
  strcpy(plugin_name,"");
  strcpy(rest_parms,"");
  sscanf(line,"%s %s %s",keyword, sname, plugin_name);

  //- get rest parameters all in one string
  char *tok;
  tok=strtok(line, " "); //- skip keyword
  tok=strtok(NULL, " "); //- skip object name
  tok=strtok(NULL, " "); //- skip plugin name
  tok=strtok(NULL, ""); //- rest of the line until newline
  if(tok) {strncpy(rest_parms, tok, sizeof(rest_parms)); }

  if(strcmp(keyword,"SECTOR")==0) {

    fgets(line,linesize,fp);
    sscanf(line, "%d %d", &numv, &nump);

    l3d_sector *sector;
    int new_onum;
    objects[new_onum=objects.next_index()] = sector = new l3d_sector(numv);
    current_sector = sector;

    strcpy(objects[new_onum]->name, sname);

    for(int v=0; v<numv; v++) {
      int vn;
      float vx, vy, vz;

      fgets(line, linesize, fp);
      sscanf(line, "%d %f %f %f", &vn, &vx, &vy, &vz);
      (*(objects[new_onum]->vertices))[vn].original.set(float_to_l3d_real(vx),
          float_to_l3d_real(vy),
          float_to_l3d_real(vz),
          int_to_l3d_real(1));
    }

    for(int p=0; p<nump; p++) {
      int p_idx = objects[new_onum]->polygons.next_index();
```

```
char *s;
fgets(line, linesize, fp);
s = strtok(line, " ");
if(strcmp(s,"GEOMPOLY")==0) {
  int num_vtx;
  l3d_polygon_3d_textured_lightmapped *pt;

  s = strtok(NULL, " ");
  sscanf(s, "%d", &num_vtx);
  objects[new_onum]->polygons[p_idx] = pt =
           new l3d_polygon_3d_textured_lightmapped(num_vtx,scache);
  objects[new_onum]->polygons[p_idx]->vlist = &objects[new_onum]->vertices;

  for(int vtx=0; vtx<num_vtx; vtx++) {
    s = strtok(NULL, " ");
    int iv;
    sscanf(s, "%d", &iv);
    (*(objects[new_onum]->polygons[p_idx]->ivertices))[
      objects[new_onum]->polygons[p_idx]->ivertices->next_index()].ivertex = iv;

  }

  pt->fovx = &(camera->fovx);
  pt->fovy = &(camera->fovy);
  pt->screen_xsize = &(screen->xsize);
  pt->screen_ysize = &(screen->ysize);

  int tex_id = 0;
  pt->texture = new l3d_texture;
  pt->texture->tex_data = & tex_data[tex_id];
  pt->texture->owns_tex_data = false;

  pt->texture->O.original.set
  (

    (**pt->vlist)[ (*pt->ivertices)[0].ivertex ].original.X_,
    (**pt->vlist)[ (*pt->ivertices)[0].ivertex ].original.Y_,
    (**pt->vlist)[ (*pt->ivertices)[0].ivertex ].original.Z_,
    int_to_l3d_real(1) );
  pt->texture->U.original.set
  (
    (**pt->vlist)[ (*pt->ivertices)[1].ivertex ].original.X_,
    (**pt->vlist)[ (*pt->ivertices)[1].ivertex ].original.Y_,
    (**pt->vlist)[ (*pt->ivertices)[1].ivertex ].original.Z_,
    int_to_l3d_real(1) );

  l3d_point V_point(
    (**pt->vlist)[ (*pt->ivertices)[pt->ivertices->num_items-1].ivertex ].original.X_,
    (**pt->vlist)[ (*pt->ivertices)[pt->ivertices->num_items-1].ivertex ].original.Y_,
    (**pt->vlist)[ (*pt->ivertices)[pt->ivertices->num_items-1].ivertex ].original.Z_,
    int_to_l3d_real(1) );
  l3d_vector
  U_vector = pt->texture->U.original - pt->texture->O.original,
             V_vector = V_point - pt->texture->O.original,
                        U_cross_V_vector = cross(U_vector, V_vector),
                               V_vector_new = normalized(cross(U_cross_V_vector, U_vector)) *
                                              sqrt(dot(U_vector,U_vector));

  pt->texture->V.original = V_vector_new + pt->texture->O.original;
```

TEAM-FLY

```
    objects[new_onum]->polygons[p_idx]->compute_center();
    objects[new_onum]->polygons[p_idx]->compute_sfcnormal();

    pt->compute_surface_orientation_and_size();

    pt->plane.align_with_point_normal(pt->center.original, normalized(pt->sfcnormal.original
              - pt->center.original));

  }else if(strcmp(s,"GEOMPOLY_TEX")==0) {

    int num_vtx;
    l3d_polygon_3d_textured_lightmapped *pt;

    s = strtok(NULL, " ");
    sscanf(s, "%d", &num_vtx);
    objects[new_onum]->polygons[p_idx] = pt = new l3d_polygon_3d_textured_lightmapped(num_vtx,
              scache);
    objects[new_onum]->polygons[p_idx]->vlist = &objects[new_onum]->vertices;

    for(int vtx=0; vtx<num_vtx; vtx++) {
      s = strtok(NULL, " ");
      int iv;
      sscanf(s, "%d", &iv);
      (*(objects[new_onum]->polygons[p_idx]->ivertices))[
        objects[new_onum]->polygons[p_idx]->ivertices->next_index()].ivertex = iv;
    }
    s = strtok(NULL, " ");

    pt->fovx = &(camera->fovx);
    pt->fovy = &(camera->fovy);
    pt->screen_xsize = &(screen->xsize);
    pt->screen_ysize = &(screen->ysize);

    int tex_id;
    sscanf(s, "%d", &tex_id);
    pt->texture = new l3d_texture;
    pt->texture->tex_data = & tex_data[tex_id];
    pt->texture->owns_tex_data = false;
    float x,y,z;
    s = strtok(NULL, " ");
    sscanf(s,"%f", &x);
    s = strtok(NULL, " ");
    sscanf(s,"%f", &y);
    s = strtok(NULL, " ");
    sscanf(s,"%f", &z);

    pt->texture->O.original.set(float_to_l3d_real(x),
                                float_to_l3d_real(y),
                                float_to_l3d_real(z),
                                int_to_l3d_real(1));
    s = strtok(NULL, " ");
    sscanf(s,"%f", &x);
    s = strtok(NULL, " ");
    sscanf(s,"%f", &y);
    s = strtok(NULL, " ");
    sscanf(s,"%f", &z);
    pt->texture->U.original.set(float_to_l3d_real(x),
                                float_to_l3d_real(y),
                                float_to_l3d_real(z),
                                int_to_l3d_real(1));
```

```
      s = strtok(NULL, " ");
      sscanf(s,"%f", &x);
      s = strtok(NULL, " ");
      sscanf(s,"%f", &y);
      s = strtok(NULL, " ");
      sscanf(s,"%f", &z);
      pt->texture->V.original.set(float_to_l3d_real(x),
                                  float_to_l3d_real(y),
                                  float_to_l3d_real(z),
                                  int_to_l3d_real(1));

      objects[new_onum]->polygons[p_idx]->compute_center();
      objects[new_onum]->polygons[p_idx]->compute_sfcnormal();

      pt->compute_surface_orientation_and_size();

      pt->plane.align_with_point_normal(pt->center.original, normalized(pt->sfcnormal.original
                - pt->center.original));

    }else {
      int num_vtx;

      s = strtok(NULL, " ");
      sscanf(s, "%d", &num_vtx);
      objects[new_onum]->polygons[p_idx] = new l3d_polygon_3d_portal(num_vtx);
      objects[new_onum]->polygons[p_idx]->vlist = &objects[new_onum]->vertices;
      for(int vtx=0; vtx<num_vtx; vtx++) {
        s = strtok(NULL, " ");
        int iv;
        sscanf(s, "%d", &iv);
        (*(objects[new_onum]->polygons[p_idx]->ivertices))[
          objects[new_onum]->polygons[p_idx]->ivertices->next_index()].ivertex = iv;
      }

      objects[new_onum]->polygons[p_idx]->compute_center();
      objects[new_onum]->polygons[p_idx]->compute_sfcnormal();

      s = strtok(NULL," ");
      char *s2 = s + strlen(s)-1;
      while(*s2=='' || *s2=='') {*s2=0; s2--; }
      l3d_polygon_3d_portal *pp;
      pp = dynamic_cast<l3d_polygon_3d_portal *>(objects[new_onum]->polygons[p_idx]);
      if(pp) {strcpy(pp->connected_sector_name, s); }
    }

  }
}else if(strcmp(keyword,"ACTOR")==0) {
  int new_onum;

  objects[new_onum = objects.next_index()]
  = (*current_sector->objects)[current_sector->objects->next_index()]
    = new l3d_object_clippable_boundable(100);
  strcpy(objects[new_onum]->name, sname);
  strcpy(objects[new_onum]->plugin_name, plugin_name);
  objects[new_onum]->parent = current_sector;

  if(strlen(plugin_name)) {
    objects[new_onum]->plugin_loader =
      factory_manager_v_0_2.plugin_loader_factory->create();
    objects[new_onum]->plugin_loader->load(plugin_name);
```

```
      objects[new_onum]->plugin_constructor =
        (void (*)(l3d_object *, void *))
        objects[new_onum]->plugin_loader->find_symbol("constructor");
      objects[new_onum]->plugin_update =
        (void (*)(l3d_object *))
        objects[new_onum]->plugin_loader->find_symbol("update");
      objects[new_onum]->plugin_destructor =
        (void (*)(l3d_object *))
        objects[new_onum]->plugin_loader->find_symbol("destructor");
      objects[new_onum]->plugin_copy_data =
        (void (*)(const l3d_object *, l3d_object *))
        objects[new_onum]->plugin_loader->find_symbol("copy_data");

      l3d_plugin_environment *e = new l3d_plugin_environment
                                  (texture_loader, screen->sinfo, scache, rest_parms);

      if(objects[new_onum]->plugin_constructor) {
        (*objects[new_onum]->plugin_constructor) (objects[new_onum],e);
      }
    }
  }else if(strcmp(keyword,"CAMERA")==0) {

    float posx,posy,posz;
    float xaxis_x, xaxis_y, xaxis_z,
    yaxis_x, yaxis_y, yaxis_z,
    zaxis_x, zaxis_y, zaxis_z;
    char *tok;

    tok = strtok(rest_parms, " ");  if(tok) {sscanf(tok, "%f", &posx); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &posy); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &posz); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &xaxis_x); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &yaxis_x); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &zaxis_x); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &xaxis_y); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &yaxis_y); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &zaxis_y); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &xaxis_z); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &yaxis_z); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &zaxis_z); }

    this->current_sector = current_sector;
    camera->VRP.set(float_to_l3d_real(posx),
                    float_to_l3d_real(posy),
                    float_to_l3d_real(posz),
                    int_to_l3d_real(1));
    camera->calculate_viewing_xform();
  }
}

for(int ii=0; ii<objects.num_items; ii++) {
  for(int pi=0; pi<objects[ii]->polygons.num_items; pi++) {
    l3d_polygon_3d_portal *p;
    if ((p = dynamic_cast<l3d_polygon_3d_portal *>(objects[ii]->polygons[pi]))) {
      p->connected_sector = sector_by_name(p->connected_sector_name);
    }
  }
}
```

```
}

void l3d_world_portal_textured_lightmapped_obj::place_lamp
(l3d_sector *sector, l3d_polygon_3d_clippable *clipwin)
{
  static int recurselevel = 0;

  if(recurselevel > 4) return;
  recurselevel ++;
  int iObj, iFacet;

  l3d_vector facet_to_cam_vec;
  l3d_vector N;

  l3d_polygon_3d_clippable *pc;

  for(int i=0; i<sector->objects->num_items; i++) {
    (*sector->objects)[i]->init_nonculled_list();
    (*sector->objects)[i]->vertices->num_varying_items = 0;
    (*sector->objects)[i]->transform_stage = 0;

    for(register int ivtx=0; ivtx<(*sector->objects)[i]->vertices->num_fixed_items; ivtx++) {
      (*((*sector->objects)[i]->vertices))[ivtx].reset();
    }

    l3d_polygon_3d_node *n;
    n = (*sector->objects)[i]->nonculled_polygon_nodes;
    while(n) {
      n->polygon->init_clip_ivertices();
      n->polygon->init_transformed();
      n=n->next;
    }

    if ((*sector->objects)[i]->num_xforms) {
      (*sector->objects)[i]->transform((*sector->objects)[i]->modeling_xform);
    }

    if (sector->num_xforms) {
      (*sector->objects)[i]->transform(sector->modeling_xform);
    }
  }

  l3d_polygon_3d_node *n_objects=0;

  for(int i=0; i<sector->objects->num_items; i++) {

    if((*sector->objects)[i]->nonculled_polygon_nodes) {
      l3d_polygon_3d_node *n_temp;
      n_temp = (*sector->objects)[i]->nonculled_polygon_nodes;
      while(n_temp->next) {
        n_temp=n_temp->next;
      }

      n_temp->next = n_objects;
      if(n_objects) {n_objects->prev = n_temp; }
      n_objects = (*sector->objects)[i]->nonculled_polygon_nodes;

      n_objects->prev = NULL;
    }
```

```
}

sector->init_nonculled_list();
sector->vertices->num_varying_items = 0;
sector->transform_stage = 0;
for(register int ivtx=0; ivtx<sector->vertices->num_fixed_items; ivtx++) {
  (*(sector->vertices))[ivtx].reset();
}

l3d_polygon_3d_node *n;
n = sector->nonculled_polygon_nodes;
while(n) {
  n->polygon->init_clip_ivertices();
  n->polygon->init_transformed();
  n=n->next;
}

if (sector->num_xforms) {
  sector->transform(sector->modeling_xform);
}

l3d_polygon_3d_node *last_sector_node;
{
  l3d_polygon_3d_node *n;

  n = n_objects;
  while(n) {
    n->polygon->set_transform_stage(sector->transform_stage);
    n = n->next;
  }

  n = sector->nonculled_polygon_nodes;
  if(n) {
    while(n->next) {n = n->next; }
    last_sector_node = n;
    n->next = n_objects;
    if(n_objects) {n_objects->prev = n; }
  }else {
    last_sector_node=NULL;
    sector->nonculled_polygon_nodes = n_objects;
  }
}

sector->camera_transform(camera->viewing_xform);

if(clipwin) {

  l3d_point origin(0,0,0,1), p0, p1;
  l3d_plane clip_plane;

  int survived=1;
  int idx0,idx1;

  idx0 = clipwin->clip_ivertices->num_items-1;
  idx1 = 0;

  while(1) {
    p1 = (**(clipwin->vlist))[(*clipwin->clip_ivertices)[idx0].ivertex].transformed;
    p0 = (**(clipwin->vlist))[(*clipwin->clip_ivertices)[idx1].ivertex].transformed;
```

```
    clip_plane.align_with_points( origin, p0, p1);

    sector->clip_to_plane(clip_plane);

    idx0 = idx1;
    if(idx0==clipwin->clip_ivertices->num_items-1) break;
    idx1++;
  }}

n = sector->nonculled_polygon_nodes;
while(n) {
  N.set(n->polygon->sfcnormal.transformed.X_ - n->polygon->center.transformed.X_,
        n->polygon->sfcnormal.transformed.Y_ - n->polygon->center.transformed.Y_,
        n->polygon->sfcnormal.transformed.Z_ - n->polygon->center.transformed.Z_,
        int_to_l3d_real(0));

  facet_to_cam_vec.set (0 - n->polygon->center.transformed.X_,
                        0 - n->polygon->center.transformed.Y_,
                        0 - n->polygon->center.transformed.Z_,
                        int_to_l3d_real(0));

  /* dot product explicitly formulated so no fixed point overflow */
  /* (facet_to_cam_vec is very large) */
  /* (use of the inline function "dot" uses default fixed precision) */
  /* equivalent to: if(N.dot(facet_to_cam_vec) > 0 ) */
  if(    (dot(N,facet_to_cam_vec) > 0 ) )
  {
  }else {
    if(n->prev) {
      n->prev->next = n->next;
    }else {
      sector->nonculled_polygon_nodes = n->next;
    }

    if(n->next) {
      n->next->prev = n->prev;
    }
  }

  n = n->next;
}

n = sector->nonculled_polygon_nodes;
while(n) {

  l3d_polygon_3d_textured_lightmapped *pt;
  l3d_polygon_3d_portal *p;

  if ( (p=dynamic_cast<l3d_polygon_3d_portal *>(n->polygon)) ) {
    place_lamp(p->connected_sector, p);
  }else if ((pt = dynamic_cast<l3d_polygon_3d_textured_lightmapped *>
                  (n->polygon)))
  {
    l3d_screen_info *si = new l3d_screen_info_indexed(MAX_LIGHT_LEVELS,
                          0, 0, 0);
    si->p_screenbuf = pt->lightmap->tex_data->data;

    l3d_rasterizer_2d_sw_lighter_imp *rast_imp = new
        l3d_rasterizer_2d_sw_lighter_imp(pt->lightmap->tex_data->width,
                                         pt->lightmap->tex_data->height,si);
```

```
      l3d_rasterizer_2d rast(rast_imp);

      rast_imp->O = pt->surface_orientation.O.transformed;
      rast_imp->U = pt->surface_orientation.U.transformed - rast_imp->O;
      rast_imp->V = pt->surface_orientation.V.transformed - rast_imp->O;

      l3d_polygon_2d_flatshaded *lightmap_poly;
      l3d_two_part_list<l3d_coordinate> *vlist;
      vlist = new l3d_two_part_list<l3d_coordinate>
              (pt->clip_ivertices->num_items);

      int i;
      lightmap_poly = new l3d_polygon_2d_flatshaded
                      (pt->clip_ivertices->num_items);
      lightmap_poly->vlist = &vlist;
      for(i=0; i< pt->clip_ivertices->num_items; i++) {
        l3d_point tex_coord;
        tex_coord =
          world_to_tex_matrix(pt->surface_orientation)
          | camera->inverse_viewing_xform
          | (**pt->vlist) [ (*pt->clip_ivertices)[i].ivertex ].transformed;

        if(tex_coord.W_ != int_to_l3d_real(1)) {
          l3d_real invw = l3d_divrr(int_to_l3d_real(1), tex_coord.W_);
          tex_coord.X_ = l3d_mulrr(tex_coord.X_, invw);
          tex_coord.Y_ = l3d_mulrr(tex_coord.Y_, invw);
          tex_coord.Z_ = l3d_mulrr(tex_coord.Z_, invw);
          tex_coord.W_ = int_to_l3d_real(1);
        }

        l3d_real ul,vl;
        ul = l3d_mulri(tex_coord.X_, pt->lightmap->tex_data->width);
        vl = l3d_mulri(tex_coord.Y_, pt->lightmap->tex_data->height);

        l3d_coordinate vtx;
        vtx.transformed.set
        (ul, vl,
         int_to_l3d_real(0),
         int_to_l3d_real(1));

        (*vlist)[i] = vtx;
        (*lightmap_poly->clip_ivertices)
        [lightmap_poly->clip_ivertices->next_index()].ivertex =
          i;
      }

      lightmap_poly->final_color = 255;
      lightmap_poly->draw(&rast);

      delete vlist;
      delete lightmap_poly;
      delete si;
      delete rast_imp;

    }
     n = n->next;
  }

  recurselevel--;
}
```

Member variable `current_sector` is a pointer to the current sector object, since the current sector is always the starting point for portal rendering. Notice that we don't need to declare a list of sectors; sectors are derived from 3D objects, and the world class already has a list of objects.

Method `sector_by_name` searches for a particular sector in the world's object list, and returns a pointer to it. This method is used to link a portal with an actual sector object when only the sector's name is known; this is the case when loading portal definitions from a file.

Overridden method `update_all` does two things: it calls the ancestor `update_all` routine, and it searches to find which sector the camera is currently in; since the camera can move, this information can change from frame to frame. The call to the ancestor `update_all` routine calls update for all objects within the object list; since each object in the list is a sector, this calls update for each sector, which in turn calls update on all of the contained objects in the sector. Thus, for each frame, the entire world is updated. Instead of updating the entire world, we could choose to call `update` only for the current sector and a few adjoining sectors, to save processing time.

The other thing `update_all` does is to search for the sector containing the camera. The search encompasses only the current sector and all immediately adjoining sectors, and takes place in camera coordinates. We check for containment of the point (0,0,0.5), which is 0.5 units in front of the camera in camera coordinates, within each sector; the sector (in camera coordinates) containing this point is the current sector. The slight displacement is the pseudo-location to avoid the portal clipping problem, as discussed earlier.

Method `draw_all` starts the portal rendering with a single method invocation; it merely asks the current sector to render itself. This then recursively causes the rendering of other sectors which are visible through the portals.

Method `place_lamp` effectively places a light at the current camera position. This method calculates a new light intensity for all visible lumels in all light maps. It depends on a light map rasterizer class, covered in the next section. The light is computed as coming from the location of the camera. The light computation progresses recursively through the portals, clipping the seen polygons to the portals. This leads to the generation of shadows, as we discussed earlier. The current camera location is not explicitly saved as a "lamp position" which can be later accessed; instead, the information is merely implicitly saved in terms of the change in the light maps themselves. A more comprehensive approach could save a list of light objects with the world, which could be dynamically created, destroyed, or moved.

Member variables `texture_loader`, `tex_data`, and `scache` manage the texture data for the world. `texture_loader` loads a texture file from disk; `tex_data` is a list of texture data objects which are shared among possibly several polygons, and `scache` is the surface cache which manages the combined texture and light map data. Method `reset_surface_cache` deletes all surfaces in the surface cache, forcing the surfaces to be re-created, by newly combining the texture and light map information. This is useful if the light map information has changed after an invocation of `place_lamp`.

The `load_from_file` routine reads portal and sector definitions from a world file. This makes testing easier; we don't have to hard code creation of portals and sectors in our program, but can instead use a data file to specify our portal world. We can also use a 3D editor such as Blender to interactively model our portal worlds. Doing so, however, requires the support of a few custom

tools and a bit of post-processing work; Chapter 8 describes the process in detail. Admittedly, the `load_from_file` routine is a bit monolithic; a better approach would read an arbitrary class name from a file, pass this class name on to an object factory which creates an instance of a streamable object, then call a virtual `load` method on the object to allow the object to load itself from the stream. A comprehensive approach would even allow for arbitrary polymorphic references to objects or lists of objects not yet loaded from disk, meaning that the entire world is not loaded at once, but only in pieces as it is needed. We don't have the time to go into such topics here, and our world file definition is very simple, so the `load_from_file` method takes the simple, function-oriented, one-function-knows-it-all approach.

The format of a world file is as follows.

```
1 TEXTURES
stone.ppm
4 SECTORS_AND_ACTORS
SECTOR main
numverts numpolys
0 x y z
1 x y z
...
11 x y z
GEOMPOLY 4 v0 v1 v2 v3
GEOMPOLY_TEX 4 v0 v1 v2 v3 tex_id ox oy oz ux uy uz vx vy vz
PORTALPOLY 4 v0 v1 v2 v3 sectorxxx
...
ACTOR act01 plugin.so x y z m00 m01 m02 m11 m10 m12 m20 m21 m22 mesh.obj mesh.ppm mesh.uv
CAMERA cam01 plugin.so x y z m00 m01 m02 m11 m10 m12 m20 m21 m22
SECTOR sector1
```

The world file begins with a line indicating the number of textures used within the file, followed by the names of the PPM texture files, one on each line. Next comes a count of sector and actor blocks in the file, followed by that same number of sector, actor, or camera blocks.

A sector block begins with the word "SECTOR" followed by the sector name. Next comes a line containing two numbers, the number of vertices and the number of polygons in the sector. Then come the vertex definitions, one per line, starting with the vertex index number, followed by the (*x*,*y*,*z*) coordinates of the vertex. Finally, we have the polygon definitions, exactly one per line.

Within a sector block, the polygon definitions use one of the keywords `GEOMPOLY`, `GEOMPOLY_TEX`, or `PORTALPOLY`. Keyword `GEOMPOLY` indicates a geometrical polygon with automatically assigned texture image and texture space. After this keyword, the first number is a count of vertex indices, followed by a clockwise specification of the integer vertex indices defining the polygon. The texture image is the first texture image in the world file, and the texture coordinates are computed assuming that the first vertex is the texture space origin, the second vertex is the tip of the *u* axis, and the last vertex is the tip of the *v* axis. Keyword `GEOMPOLY_TEX` specifies a polygon with an explicitly specified texture image and space. After this keyword, first come the vertex index count and a list of vertex indices. Then comes an integer texture identifier, starting with 0, specifying which image in the texture section should be used for this polygon. Finally come nine numbers, specifying the texture space origin, tip of the *u* axis, and tip of the *v* axis. Keyword `PORTALPOLY` indicates a portal. Following this keyword are the vertex index count and a list of vertex indices. Then comes a string specifying the name of the sector to which

the portal leads; the actual pointer to the sector object is assigned later, after all sectors have been read into memory.

An actor block in the world file consists of exactly one line starting with the keyword `ACTOR`. This specifies a dynamic object (of type `l3d_object` or one of its descendants) which will be created in the world and which is located in the immediately preceding sector in the world file. Following this keyword is the name of the object and the name of a dynamic library plug-in file controlling the behavior of the object. All following parameters are then plug-in specific, and will be parsed by the plug-in itself. Let's look at the parameters used by the Videoscape plug-in of Chapter 3. Listed first is the initial (x,y,z) position of the object, followed by nine numbers representing the rotation matrix of the object specified in row-major order (first horizontally, then vertically). Listed next are three filename specifications: the Videoscape mesh file defining the geometry for this actor, the PPM texture file for the mesh, and the (u,v) coordinates file specifying the texture coordinates for the mesh (all of which we saw in Chapter 3). The `vidmesh.so` plug-in file is responsible for extracting and using these parameters. If we specified a different plug-in on the `ACTOR` line, it could define and parse its own parameter list in the world file; all parameters after the plug-in name are simply handed off to the plug-in for further processing.

A camera block in the world file consists of exactly one line starting with the keyword `CAMERA`. This specifies the location of the camera in the world; it will begin in the immediately preceding sector in the world file. Listed next is the name of the camera object, a plug-in filename, and the (x,y,z) location of the camera and its 3 3 rotation matrix. Currently, only the location of the camera is parsed by the world class; the name, plug-in, and orientation parameters are not yet used. However, this specification scheme treats the camera similar to the way that it treats objects, meaning that a derived world class could load a plug-in to control the behavior of the camera. In the existing world class, though, the camera is treated as a user-controllable object without plug-in behavior, just as we have been doing all along.

Class l3d_rasterizer_2d_sw_lighter_imp

Class `l3d_rasterizer_2d_sw_lighter_imp` is a software polygon rasterizer implementation, which is designed to rasterize a polygon directly into the lumels in a light map (as opposed to rasterizing into the pixels in an off-screen frame buffer). The intensity of the lumels is computed during the rasterization process.

As we saw earlier, the `place_lamp` method of class `l3d_world_portal_textured_lightmapped_obj` effectively places a light at the current camera position, and traverses the portals to determine which parts of which polygons are visible to the light source. This leads to generation of shadows. To generate shadows in this way, we need to rasterize the portal-clipped polygon in lumel space. As we discussed, we first need to obtain the coordinates of the clipped polygon's vertices in lumel space, then we rasterize the polygon into the light map, computing a light intensity for each lumel that we rasterize.

Method `draw_polygon_flatshaded` draws a "flat-shaded" polygon into the rasterizer buffer, which in this case is the light map data itself. "Flat-shaded" is a bit of a misnomer, but it is the most appropriate function to override from the inherited software rasterizer implementation, since during rasterization we only keep track of edge coordinates, not texture coordinates. This

Team-Fly®

method draws the polygon into the light map, using the same logic as the flat-shaded polygon rasterizer. There are only two changes to the rasterizer logic. First, instead of assigning a fixed color to each lumel we rasterize, we call `compute_light_level` to determine what the light level should be at the current lumel. Second, the horizontal and vertical size of the polygon is artificially increased, to allow for lumels on the edge of the polygon to be included in the light map computation.

Let's briefly discuss why we need to artificially increase the size of the polygon for light map rasterization purposes. The reason for this goes all the way back to a discussion in the introductory companion book *Linux 3D Graphics Programming*, when we first developed the rationale for our rasterization strategy. There, we said that it was of primary importance to plot every pixel once and only once, so that the polygons all lined up on the edges. We used the rule that only pixels whose upper left corner was in the polygon were drawn. The problem is that using this same strategy here often leads to unsightly dark holes or dark strips in the light map, because certain lumels exactly on the edge of the polygon do not get rasterized and thus remain dark. When computing a light map, we should ideally compute a light intensity for every lumel of which any part—not just the upper-left corner—touches the polygon. Then, depending on how much of the lumel lies within the polygon, we would scale the computed intensity accordingly; a lumel only half within the polygon would only get half of its computed light intensity, for instance. This technique is called *anti-aliasing*. The shadow light map rasterizer here doesn't perform anti-aliasing, but it instead artificially expands the horizontal and vertical size of the polygon to cause it to cover more lumels. This means that the generated shadows are somewhat inaccurate, and sometimes contain a hole or two, but the code does serve its illustrative purpose in a working example.

Method `compute_light_level` computes the light level for a given 2D lumel coordinate (u,v). It first goes from light map space into world space—a very easy transformation using the axes of the light map space, as we saw in Chapter 2—then computes a light value using the diffuse reflection equation. The variables for storing the axes of the light map space are inherited from class `l3d_texture_computer`, and must have been set beforehand by the external caller.

Listing 5-14: `ras_swl.h`

```
#ifndef __RAS_SWL_H
#define __RAS_SWL_H
#include "../tool_os/memman.h"

#include "ras_sw.h"
#include "math.h"
#include "../system/sys_dep.h"
#include "../geom/texture/texture.h"

class l3d_rasterizer_2d_sw_lighter_imp :
      virtual public l3d_rasterizer_2d_sw_imp,
      virtual public l3d_texture_computer
{
  protected:
    virtual int compute_light_level(int lumel_x, int lumel_y);
  public:

    l3d_rasterizer_2d_sw_lighter_imp(int xs, int ys, l3d_screen_info *si) :
    l3d_rasterizer_2d_imp(xs,ys,si),
```

```
    l3d_rasterizer_2d_sw_imp(xs,ys,si) {};
    void draw_polygon_flatshaded(const l3d_polygon_2d_flatshaded *p_poly);
};

#endif
```

Listing 5-15: ras_swl.cc

```
#include "ras_swl.h"
#include "../system/sys_dep.h"
#include <string.h>
#include <stdlib.h>
#include "../tool_os/memman.h"

int l3d_rasterizer_2d_sw_lighter_imp::compute_light_level(int lumel_x, int lumel_y)
{
  l3d_real x,y,z;

  l3d_real u,v;

  u = l3d_divrr( int_to_l3d_real(lumel_x), xsize );
  v = l3d_divrr( int_to_l3d_real(lumel_y), ysize );

  x = O.X_ + l3d_mulri(U.X_, u) + l3d_mulri(V.X_, v);
  y = O.Y_ + l3d_mulri(U.Y_, u) + l3d_mulri(V.Y_, v);
  z = O.Z_ + l3d_mulri(U.Z_, u) + l3d_mulri(V.Z_, v);

  l3d_vector aLight(0,0,0,1);

  l3d_vector L(aLight.X_ - x ,
               aLight.Y_ - y ,
               aLight.Z_ - z ,
               int_to_l3d_real(0)),

  N(cross(U,V));

  l3d_real intensity;

  l3d_real f_att=l3d_divrr(int_to_l3d_real(1),
          (float_to_l3d_real(0.1)+float_to_l3d_real(0.01*sqrt(l3d_real_to_float(dot(L,L))))));
  if (f_att>int_to_l3d_real(1)) f_att=int_to_l3d_real(1);

  l3d_real n_dot_l;
  n_dot_l = dot(normalized(N), normalized(L));

  intensity = l3d_mulrr ( l3d_mulrr(int_to_l3d_real(128) , f_att) ,
                          n_dot_l);
  if (intensity>int_to_l3d_real(128)) {intensity = int_to_l3d_real(128); }
  if (intensity<int_to_l3d_real(2))  {intensity = int_to_l3d_real(2);  }

  return l3d_real_to_int(intensity);
}

void l3d_rasterizer_2d_sw_lighter_imp::draw_polygon_flatshaded
(const l3d_polygon_2d_flatshaded *p_poly)
{
  l3d_real x0,y0,x1,y1,x2,y2,x3,
  left_x_start,left_y_start,left_x_end,left_y_end,left_dx,left_x,leftedge,
  right_x_start,right_y_start,right_x_end,right_y_end,right_dx,right_x,rightedge;
  int left_ceily_start, left_ceily_end,
  right_ceily_start, right_ceily_end, scanline;
```

```
  l3d_real top_y, bottom_y;
  int point_on_right=0;
  int left_idx, right_idx, top_y_idx, bottom_y_idx;

  int clipleftedge, cliprightedge;
  int maxy_upper, iceil_y0, iceil_y1, iceil_y2;

  int i;

  //- convenience macro for accessing the vertex coordinates. Notice
  //- that we access the clipped vertex index list (not the original),
  //- and the transformed coordinate (not the original). This means
  //- we draw the clipped version of the transformed polygon.
#define VTX(i) ((**(p_poly->vlist))[ (*(p_poly->clip_ivertices))[i].ivertex ].transformed)

#define XPAD 4
#define YPAD 1
  //-----------------------------------------------------------------------
  //- STEP 1: Find top and bottom vertices
  //-----------------------------------------------------------------------

  top_y = VTX(0).Y_;
  top_y_idx = 0;
  bottom_y = top_y;
  bottom_y_idx = top_y_idx;
  for(i=0; i<p_poly->clip_ivertices->num_items; i++) {

    if(VTX(i).Y_ < top_y) {
      top_y = VTX(i).Y_;
      top_y_idx = i;
    }
    if(VTX(i).Y_ > bottom_y) {
      bottom_y = VTX(i).Y_;
      bottom_y_idx = i;
    }
  }

  left_idx = top_y_idx;
  right_idx = top_y_idx;

  //-----------------------------------------------------------------------
  //- STEP 2: Create empty left edge
  //-----------------------------------------------------------------------
  left_x_start = VTX(top_y_idx).X_;
  left_y_start = VTX(top_y_idx).Y_;
  left_ceily_start=iceil(left_y_start);
  left_ceily_end=left_ceily_start;

  //-----------------------------------------------------------------------
  //- STEP 3: Create empty right edge
  //-----------------------------------------------------------------------
  right_x_start=left_x_start;
  right_y_start=left_y_start;
  right_ceily_start=left_ceily_start;
  right_ceily_end=right_ceily_start;

  //-----------------------------------------------------------------------
  //- STEP 4: Loop from top y to bottom y
  //-----------------------------------------------------------------------
```

```
scanline = left_ceily_start;

int first_scanline_done = 0;
while(scanline <= ysize) {

  //-------------------------------------------------------------------------
  //- STEP 5: Find next left-edge of non-zero height (if needed)
  //-------------------------------------------------------------------------
  while( left_ceily_end - scanline <= 0 ) {
    if (left_idx == bottom_y_idx) return;  //- done
    left_idx = p_poly->next_clipidx_left
               (left_idx, p_poly->clip_ivertices->num_items);
    left_y_end= VTX(left_idx).Y_;
    left_ceily_end = iceil(left_y_end);

    if(left_ceily_end - scanline) { //- found next vertex
      //-----------------------------------------------------------------
      //- STEP 5a: Initialize left-x variables
      //-----------------------------------------------------------------
      left_x_end= VTX(left_idx).X_;
      left_dx = l3d_divrr(left_x_end-left_x_start,left_y_end-left_y_start);
      left_x = left_x_start +  //- sub-pixel correction
               l3d_mulrr(int_to_l3d_real(left_ceily_start)-left_y_start , left_dx);
    }else { //- did not find next vertex: last failed end = new start
      left_x_start = VTX(left_idx).X_;
      left_y_start = VTX(left_idx).Y_;
      left_ceily_start = iceil(left_y_start);
    }
  }

  //-------------------------------------------------------------------------
  //- STEP 6: Find next right-edge of non-zero height (if needed)
  //-------------------------------------------------------------------------
  //- if needed, find next right-edge whose ceily_end > current scanline
  //- this is a while and not an if, as desribed above...
  while(right_ceily_end - scanline <= 0 ) {
    if (right_idx == bottom_y_idx) return;  //- done
    right_idx = p_poly->next_clipidx_right
                (right_idx,p_poly->clip_ivertices->num_items);
    right_y_end=VTX(right_idx).Y_;
    right_ceily_end = iceil(right_y_end);
    if(right_ceily_end - scanline) { //- found next vertex
      //-----------------------------------------------------------------
      //- STEP 6a: Initialize right-x variables
      //-----------------------------------------------------------------
      right_x_end=VTX(right_idx).X_;
      right_dx =
        l3d_divrr(right_x_end-right_x_start,right_y_end-right_y_start);
      right_x = right_x_start +  //- sub-pixel correction
                l3d_mulrr(int_to_l3d_real(right_ceily_start)-right_y_start , right_dx);
    }else { //- did not find next vertex: last failed end = new start
      right_x_start = VTX(right_idx).X_;
      right_y_start = VTX(right_idx).Y_;
      right_ceily_start = iceil(right_y_start);
    }
  }

  //- clamp edge values to screen
  if (left_ceily_end > ysize) left_ceily_end = ysize;
```

```
if (right_ceily_end > ysize) right_ceily_end = ysize;

//-------------------------------------------------------------------
//- STEP 7: Loop until left and/or right edge is finished
//-------------------------------------------------------------------
while ( (scanline < left_ceily_end) && (scanline < right_ceily_end) ) {
  if(! first_scanline_done) {
    first_scanline_done = 1;

    int top_pad;
    top_pad = scanline -YPAD;
    if(top_pad < 0) top_pad = 0;
    int target = scanline;

    for(scanline=top_pad ; scanline < target; scanline++) {

      clipleftedge = ifloor(left_x)-XPAD;

      clipleftedge = (clipleftedge < 0) ? 0 : clipleftedge;
      cliprightedge = iceil(right_x)+XPAD;

      cliprightedge = (cliprightedge > xsize-1) ? xsize-1 : cliprightedge;

      int lumel_x = clipleftedge;
      for(register char unsigned *pix =
            address_of_point(clipleftedge,
                             SW_RAST_Y_REVERSAL(ysize,scanline));

          pix <= address_of_point(cliprightedge, scanline);

          lumel_x++

         )
      {
        int light_value = compute_light_level(lumel_x, scanline);

        light_value += *pix;
        if(light_value > 255) light_value = 255;
        draw_point_at_paddress(&pix, light_value);
      }

      left_x+= left_dx;
      right_x+= right_dx;
    }

  }

  clipleftedge = ifloor(left_x)-XPAD;

  clipleftedge = (clipleftedge < 0) ? 0 : clipleftedge;
  cliprightedge = iceil(right_x)+XPAD;

  cliprightedge = (cliprightedge > xsize-1) ? xsize-1 : cliprightedge;

  {
    //-----------------------------------------------------------------
    //- STEP 8: Draw horizontal span
    //-----------------------------------------------------------------
    int lumel_x = clipleftedge;
    for(register char unsigned *pix =
```

```
            address_of_point(clipleftedge,
                             SW_RAST_Y_REVERSAL(ysize,scanline));

          pix <= address_of_point(cliprightedge, scanline);

          lumel_x++

         )
      {
        int light_value = compute_light_level(lumel_x, scanline);

        light_value += *pix;
        if(light_value > 255) light_value = 255;
        draw_point_at_paddress(&pix, light_value);
      }
    }

    //-------------------------------------------------------------------
    //- STEP 9: Increase y, and step x-values on left and right edges
    //-------------------------------------------------------------------
    scanline++;
    left_x += left_dx;
    right_x += right_dx;
  }

  //-------------------------------------------------------------------
  //- STEP 10: Continue loop, looking for next edge of non-zero height
  //-------------------------------------------------------------------

  //- for the left and/or right segment(s) which just completed drawing
  //- initialize xxx_start = xxx_end, to begin next segment. xxx_end is
  //- then searched for in the next iteration (the while() loops)
  if ( left_ceily_end - scanline <= 0 ) {
    left_x_start=left_x_end;
    left_y_start=left_y_end;
    left_ceily_start=left_ceily_end;
  }
  if ( right_ceily_end - scanline <= 0 ) {
    right_x_start=right_x_end;
    right_y_start=right_y_end;
    right_ceily_start=right_ceily_end;
  }
}

for(int bottom_pad=0; bottom_pad<YPAD; bottom_pad++) {
  if(scanline < ysize-1) {
    scanline++;
    left_x += left_dx;
    right_x += right_dx;

    clipleftedge = ifloor(left_x)-XPAD;

    clipleftedge = (clipleftedge < 0) ? 0 : clipleftedge;
    cliprightedge = iceil(right_x)+XPAD;

    cliprightedge = (cliprightedge > xsize-1) ? xsize-1 : cliprightedge;

    int lumel_x = clipleftedge;
    for(register char unsigned *pix =
```

```
          address_of_point(clipleftedge,
                           SW_RAST_Y_REVERSAL(ysize,scanline));

        pix <= address_of_point(cliprightedge, scanline);

        lumel_x++

       )
      {
        int light_value = compute_light_level(lumel_x, scanline);

        light_value += *pix;
        if(light_value > 255) light_value = 255;
        draw_point_at_paddress(&pix, light_value);
      }

      scanline++;
      left_x++;
      right_x++;
    }
  }

}
#undef VTX(i)
```

Class l3d_pipeline_world_lightmapped

Class `l3d_pipeline_world_lightmapped` is a pipeline class handling the new keyboard input required by light mapped portal world objects. The two new keys are p and r. Press **p** to place a lamp at the current camera position. Press **r** to reset the surface cache, causing the newly computed light maps to be displayed.

Listing 5-16: `pi_lwor.h`

```
#ifndef __PI_LWOR_H
#define __PI_LWOR_H
#include "../tool_os/memman.h"

#include "pi_wor.h"
#include "../geom/world/world.h"

class l3d_pipeline_world_lightmapped : public l3d_pipeline_world {
  public:
    l3d_pipeline_world_lightmapped(l3d_world *w) :
    l3d_pipeline_world(w) {};
    void key_event(int ch);
};

#endif
```

Listing 5-17: `pi_lwor.cc`

```
#include "pi_wor.h"

#include <sys/time.h>
#include <unistd.h>
#include <string.h>
#include "../tool_os/memman.h"

void l3d_pipeline_world::key_event(int c) {
  switch(c) {
```

```
    case 'q': done = 1;

      //- default keys for world->camera movement -- full 360 degree movement
    case 'j': if(world->camera->rot_VUP_velocity>int_to_l3d_real(-95)) world->camera->rot_VUP_velocity
-= int_to_l3d_real(5); break;
    case 'l': if(world->camera->rot_VUP_velocity<int_to_l3d_real( 95)) world->camera->rot_VUP_velocity
+= int_to_l3d_real(5); break;
    case 'J': if(world->camera->rot_VFW_velocity<int_to_l3d_real( 95)) world->camera->rot_VFW_velocity
+= int_to_l3d_real(5); break;
    case 'L': if(world->camera->rot_VFW_velocity>int_to_l3d_real(-95)) world->camera->rot_VFW_velocity
-= int_to_l3d_real(5); break;
    case 'I': if(world->camera->rot_VRI_velocity<int_to_l3d_real( 95)) world->camera->rot_VRI_velocity
+= int_to_l3d_real(5); break;
    case 'K': if(world->camera->rot_VRI_velocity>int_to_l3d_real(-95)) world->camera->rot_VRI_velocity
-= int_to_l3d_real(5); break;
    case 'k': if(world->camera->VFW_velocity    >int_to_l3d_real(-90)) world->camera->VFW_velocity -=
world->camera->VFW_thrust; break;
    case 'i': if(world->camera->VFW_velocity    <int_to_l3d_real( 90)) world->camera->VFW_velocity +=
world->camera->VFW_thrust; break;
    case 's': if(world->camera->VRI_velocity    >int_to_l3d_real(-90)) world->camera->VRI_velocity -=
world->camera->VRI_thrust; break;
    case 'f': if(world->camera->VRI_velocity    <int_to_l3d_real( 90)) world->camera->VRI_velocity +=
world->camera->VRI_thrust; break;
    case 'd': if(world->camera->VUP_velocity    >int_to_l3d_real(-90)) world->camera->VUP_velocity -=
world->camera->VUP_thrust; break;
    case 'e': if(world->camera->VUP_velocity    <int_to_l3d_real( 90)) world->camera->VUP_velocity +=
world->camera->VUP_thrust; break;

      //- field-of-view modification
    case 'X': world->camera->fovx = world->camera->fovx + float_to_l3d_real(0.1); break;
    case 'x': world->camera->fovx = world->camera->fovx - float_to_l3d_real(0.1); break;
    case 'Y': world->camera->fovy = world->camera->fovy + float_to_l3d_real(0.1); break;
    case 'y': world->camera->fovy = world->camera->fovy - float_to_l3d_real(0.1); break;

      //- speed
  case 'v': case 'V':
      world->camera->VUP_thrust += int_to_l3d_real(1.0);
      if(world->camera->VUP_thrust  > 3.0) {
        world->camera->VUP_thrust  -= int_to_l3d_real(3.0);
      }
      world->camera->VFW_thrust = world->camera->VUP_thrust;
      world->camera->VRI_thrust = world->camera->VUP_thrust;
      break;

      //- display

  case 'n': case 'N':
      world->should_display_status = 1-world->should_display_status; //- toggle
      break;

  }

}

void l3d_pipeline_world::update_event() {
  world->update_all();
}

void l3d_pipeline_world::draw_event(void) {
  static int frame=0;
```

```
  static long last_sec, last_usec;
  long sec, usec;
  static float fps;
  static char str[80];

  struct timeval tv;
  struct timezone tz;
  tz.tz_minuteswest = -60;

  world->rasterizer->clear_buffer();
  world->draw_all();

  frame++;
  if((frame & 0x3F) == 0) {
    gettimeofday(&tv,&tz);
    fps = 64 * 1.0 / ( (tv.tv_sec + tv.tv_usec/1000000.0) - (last_sec + last_usec/1000000.0) );
    last_sec = tv.tv_sec;
    last_usec = tv.tv_usec;

  }

  if(world->should_display_status) {
    sprintf(str,"FPS: %f", fps);
    world->rasterizer->draw_text(0, 0, str);
  }

  world->screen->blit_screen();

}
```

Sample Program: porlotex

The last sample program in this chapter, `porlotex`, is a simple example using the above portal classes to create an indoor portal world, complete with objects, lights, and shadows. With the above classes for handling portals and sectors, the main program merely needs to create an instance of the new portal-based world class, and ask the world class to load the world database from a file, in this case file `world.dat`.

The floor plan for `world.dat` appears in Figure 5-28. After starting the program, you will be located at the camera location indicated in the figure. Initially, the scene will be fairly dark. Press **p** to place a lamp at the current camera position; press **r** to reset the surface cache and see the newly computed light maps. From the starting position, try placing a few lamps at the current camera position, then move into the larger room ahead, to see the shadowing effect of portal light map computation. Also notice that we have placed three Videoscape plug-in objects in the room at the lower left of the floor plan. Their meshes are loaded from the file `torch.obj`. Support for arbitrary objects within a sector means we are free to place props within the room to arrange our scene, much as a stage designer does.

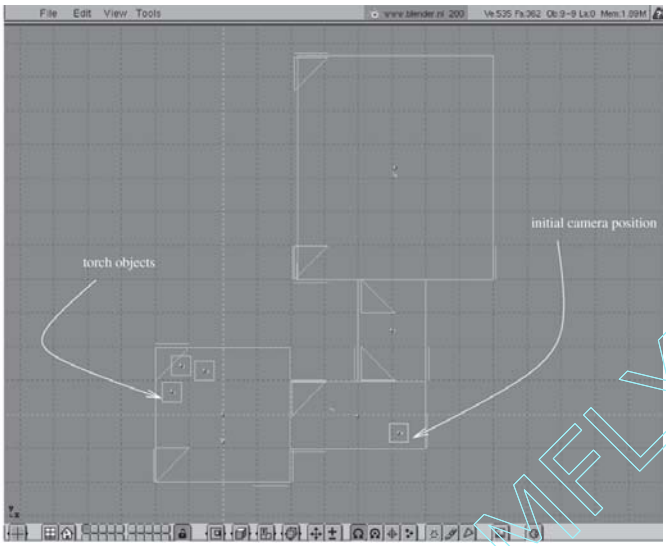


Figure 5-28: The floor plan for the world database for program `porlotex`.

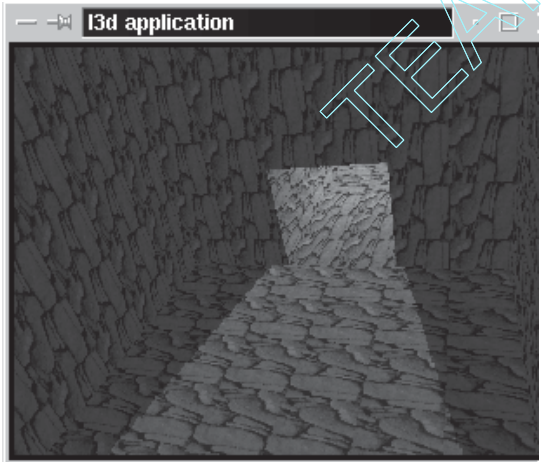


Figure 5-29: Shadows generated by the portals in program `porlotex`.

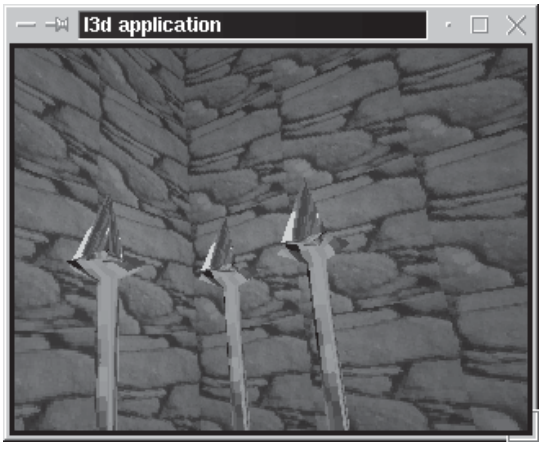


Figure 5-30: Program `porlotex` *also can load arbitrary meshes, such as these torches, into the portal world.*

TEAMFLY

The ASCII contents of the `world.dat` text file define the sectors, portals, polygons, and objects within the world, according to the world format described earlier. Since it is an ASCII text file, you could create this file by hand, but this is unbearably tedious for all but the simplest of worlds. A more realistic approach is to use a 3D modeling program to create the world file for us; see Chapter 8.

Listing 5-18: `main.cc`, main program file for program `porlotex`

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../lib/geom/object/object3d.h"
#include "../lib/geom/polygon/p_flat.h"
#include "../lib/geom/polygon/p_portal.h"
#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rasteriz.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/geom/world/world.h"
#include "../lib/system/fact0_3.h"
#include "../lib/geom/texture/texload.h"
#include "../lib/geom/texture/tl_ppm.h"
#include "../lib/pipeline/pi_lwor.h"
#include "../lib/geom/world/w_porlotex.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

main() {
  factory_manager_v_0_2.choose_factories();

  l3d_dispatcher *d;
  l3d_pipeline_world_lightmapped *p;
  l3d_world_portal_textured_lightmapped_obj w(320,240);
  void *startinfo;

  w.load_from_file("world.dat");
  w.camera->near_z = float_to_l3d_real(0.5);
  w.camera->far_z = int_to_l3d_real(5000);

  d = factory_manager_v_0_2.dispatcher_factory->create();

  p  = new l3d_pipeline_world_lightmapped(&w);
  d->pipeline = p;
  d->event_source = w.screen;

  d->start();

  delete p;
  delete d;
}
```

Other VSD Algorithms

There are many other VSD algorithms. Recently, there has been a bit of interest in occlusion-based algorithms. The main idea behind such algorithms is that if we can find large polygons near the camera, they will obscure or *occlude* polygons behind them. Assuming, as is usually the case, that the polygons are opaque and not transparent, then finding large occluder polygons early on means that we do not need to draw or process anything behind the occluder. It should be noted that while the *z* buffer essentially is an occlusion algorithm, it operates at the pixel level, and requires us to compare each pixel of each object to be rasterized with each pixel in the *z* buffer. Only after all pixels of all objects have been compared against the *z* buffer is the image ready for display. In contrast to the *z* buffer, occlusion algorithms try to save work by completely eliminating any processing of polygons which lie completely behind occluders.

Some occlusion culling algorithms rely on shadow volumes, where a large occluder acts like a portal polygon in reverse. We find an occluder polygon (generally a dynamic search and not a static specification as with portals) and create a frustum; then, we discard everything inside the frustum, because it lies behind the occluder. The difficulty here lies in finding good occluders quickly at run time. Good occluders are those which are near the camera and cover a large amount of screen space for the current camera position. We can preprocess the world database to store "large" polygons in a special potential occluder list, to limit the search space at run time. Another interesting occlusion algorithm is the hierarchical occlusion map, or HOM [ZHAN97], which creates a hierarchy of *occlusion maps*, from low resolution to high resolution, for each potential occluder. An occlusion map is something like a 2D transparency map: where it is opaque, nothing behind that pixel in the occlusion map can be seen; where it is transparent or semi-transparent, something behind that pixel might be seen. In the middle of an opaque polygon, the occlusion map is completely opaque; exactly on the edge, the occlusion map is semi-transparent. The hierarchy allows us to zoom in on semi-transparent regions where needed, whereas completely opaque regions are known to occlude their background. The occlusion idea is similar to the shadow volume method; the main difference is that the hierarchy of occlusion maps allows for incremental and approximate determination of occlusion by successive opacity comparisons, whereas the shadow volume method (as described above) computes an exact frustum for the exact outline of the occluder.

There is also a hierarchical version of the *z* buffer [MOEL99, GREE93], which can also be viewed as an occlusion algorithm similar to the *z* buffer, but hierarchical to avoid always comparing every pixel against the *z* buffer for occlusion. With this algorithm, we store and update a hierarchy of *z* buffers, all simultaneously. The *z* buffer hierarchy progresses from low resolution to high resolution, where each pixel in a lower resolution *z* buffer stores the maximum *z* value of a 4 4 sample of the next higher resolution *z* buffer. In this way, each *z* value in a lower resolution *z* buffer is a maximum (farthest) *z* coverage value of a particular region of the highest resolution *z* buffer, beyond which any primitives are invisible. If we combine this with an octree spatial partitioning method, then we can cull octree nodes quickly against the *z* buffers by comparing the depth of their faces to the depth in the coarser levels in the *z* buffer hierarchy. We traverse the hierarchy

from coarser levels to finer levels. If at any level we find that the octree node's faces lie completely behind the value in the *z* buffer at that level in the hierarchy, we know that face lies behind every primitive (even lower in the hierarchy) and thus is occluded. One problem with this algorithm is that updating a hierarchy of *z* buffers takes considerable time, probably too much time with a software-only implementation of the *z* buffer hierarchy.

One last VSD algorithm which deserves mention is ray tracing. It is currently far too slow for real-time use, but in time (decades, probably) it will eventually become viable for interactive applications. With ray tracing, we mathematically trace a 3D ray from the eye through each pixel in the screen. Then, for each ray, we see which object in the world is the first one to intersect the ray. This object is the one which is then visible for the pixel pierced by the ray. So, we compute a color value for the object at its point of intersection with the ray (with lighting and texture mapping, for instance), and draw the resultant color in the pixel. We are essentially casting "reverse" rays of light from the eye out to the scene to see which objects they hit, reversing the physical model of light where rays come from the objects to the eye. In a way, ray tracing is similar to texture mapping, where we also traced a ray from the eye to the *u*-*v* plane of texture space, but with ray tracing, we have to intersect the ray with every object in the scene to see which one is the first along the ray, to solve the VSD correctness problem. With ray tracing, we also can recursively trace rays from object to object, allowing the ray to bounce off of a surface. This simulates reflection and even refraction of light, making ray tracing a very realistic, but very time-consuming VSD algorithm. For instance, with a 640 480 screen, with just ten objects in the scene, we need to perform 3,072,000 intersection operations for every frame, assuming no recursive ray tracing (i.e., no reflection or refraction of rays). Of course, partitioning and hierarchy can be used to reduce the amount of work, but the sheer number of intersection calculations needed by ray tracing will probably limit the use of ray tracing to non-interactive applications, for some time to come.

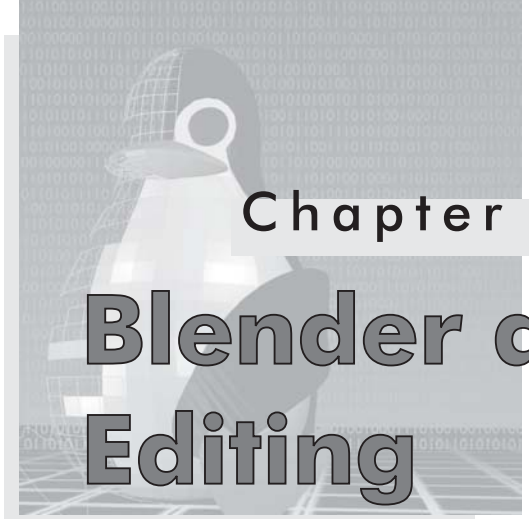
Summary

This chapter discussed VSD algorithms based on some sort of space partitioning. We looked at BSP trees, octrees, and portals. We saw how convexity prevents self-occlusion and thus enables simple rendering with back-face culling and portals. We discussed how a BSP tree can create a hierarchy of convex regions automatically. We also looked at several sample programs illustrating these techniques, culminating in a portal-based example complete with plug-in objects, texture mapping, light, and shadows.

We've now seen a number of VSD techniques which not only ensure a correct display, but which also are useful for creating larger interactive 3D environments. In general, we can use the techniques of partitioning, hierarchy, and minimizing search space in order to handle larger 3D worlds.

We now know some techniques for handling larger worlds, but how do we create these larger worlds? Although the world file is a text file, for more complex worlds, we can't type in all of our vertices, polygons, portals, sectors, and objects into the world file; it would take too long. We need to use a tool to create our worlds. In Chapter 3, we used Blender to create individual models for

import. Now, we are ready to use Blender to create the worlds themselves in which these objects are located. This is a conceptually simple idea, but requires quite a bit of work and some somewhat unusual tricks to actually obtain a working system. Chapter 6 shows you one way of using Blender to create portal worlds for import into l3d programs.



Chapter 6

Blender and World Editing

The creation of any kind of complex information structure generally requires the help of an external tool. Even with a well-defined format for how the information must look, there is always a certain complexity level above which it becomes impractical to work with the information at the lowest, raw level. Tools help by combining raw data into more meaningful, higher-level abstractions which the human brain can more easily manage.

Creating 3D worlds is a perfect example of how good tools can raise the abstraction level of the work to be done. In Chapter 5, we defined a simple ASCII world file format which allowed the specification of 3D worlds as a combination of sectors, portals, and arbitrary, autonomous polygonal objects inhabiting these sectors. Since the definition file is ASCII, it can be read and created with a simple text editor. Indeed, doing so is a valuable exercise to ensure that you understand exactly how vertices are specified, how polygons are constructed as clockwise lists of indices into this vertex list, and how objects are placed and oriented in 3D space. The main skill you must possess for such low-level 3D data manipulation is the ability to spatially visualize the location of a point just given its coordinates.

But rising world complexity places a limit on the types of 3D worlds that humans can produce directly in ASCII. It's much more convenient for us to work at a higher and visual level, visually placing objects where they should be located in the world, instead of mathematically specifying their coordinates.

In this chapter, we'll take a look at how we can use Blender to create worlds for use in the portal engine developed in Chapter 5. First, we look at some general concepts concerning world editing, then look at one way of using Blender to achieve our goals. Specifically, we cover:

- General world editing ideas and options
- Requirements of a portal world editing system
- Blender techniques to create portal worlds
- Using external tool `blend_at` to associate texture and attribute information with meshes
- Using external tool `vid2port.pl` to convert Videoscape meshes into the portal world file of Chapter 5
- Other possible world editing systems for other world partitioning schemes.

It's important to note that while the majority of this chapter focuses on one possible way of using Blender to create portal worlds, there are a number of different ways to create a world editing

system with Blender or another 3D modeling package. The last part of this chapter discusses some alternatives, so that you can make a world editing system best suited to the worlds you wish to create.

World Editing

We use the term *world editing* to describe the process of creating a specification of several 3D objects which together form a 3D world. The specification is typically in the form of a data file, which we call the *world file*. (Instead of being a file, the world specification could also be a relational or object-oriented database, for instance.) A *world editor* is a program that allows us to create a world file in a visual manner, graphically placing objects at their desired locations in 3D.

Often, the terms *level editing*, *level file*, and *level editor* are used instead of the world-based terms. This terminology comes from the computer game industry, where each level forms a particular phase of the game which must be completed. Each separate level is typically viewed as a separate unit of abstraction and is stored in its own file, thus making the level the unit of work.

For our purposes, we want to find a world editing solution that allows us to create world files corresponding to the format defined in Chapter 5. However, the ideas and techniques presented in this chapter also apply to world editing in general. For instance, a project is currently underway, using the techniques developed in this chapter, to create a level editing system for the Linux version of the World Foundry game development system (see Chapter 9). You could also use the same ideas to create a VRML development environment, allowing you to place objects and behaviors within a 3D scene designed for viewing and interaction through a WWW browser. Thus, while our immediate goal is the visual creation of a world file having the format described in Chapter 5, you should keep in mind the broader world editing context in which the ideas of this chapter can be understood.

Now, let's look at the types of world editing solutions available to us.

No World Editor

The simplest world editing solution is to use no world editor. This means directly editing the world file in ASCII. As mentioned before, this is technically possible, and for debugging purposes, sometimes necessary. Without a world editor, you need a very good mental visualization capability, so that you can imagine what an object looks like and where it is located, just by looking at its ASCII coordinate and face definitions. This visualization capability comes in handy when debugging 3D code, since all you have in the debugger are the raw coordinates; you have to determine by inspection which values seem incorrect or illogical in order to find the error.

Creating worlds without a world editor can be done for simple, box-like geometry, but even for simple worlds, you probably need to sketch out the world on paper to keep track of the coordinate and face definitions.

If you develop a custom 3D program or library, world editing is probably not the highest priority in the early stages of development; more important at this stage are the core architecture and basic operation of the system. Thus, it is entirely conceivable and even natural to work without a

world editor in the beginning. After a while, though, the great effort needed to make even small changes to the world files becomes a nuisance, at which point it makes more sense to use a visual world editing program.

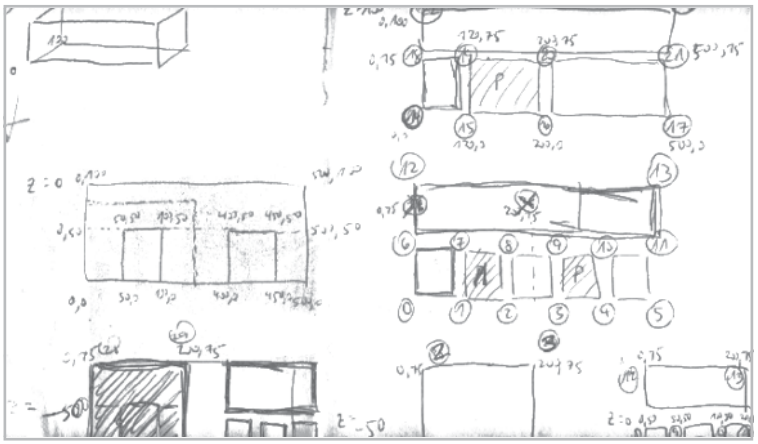


Figure 6-1: Having no world editor means keeping track of coordinates on paper and in your head.

Write Your Own World Editor

Writing a custom world editor allows you to create a visual world editing system exactly tailored to your 3D application. Such a world editing system can be programmed to allow or disallow certain combinations of objects depending on the underlying 3D library. For instance, if you want to limit the number of objects per sector to be 10, then you could disallow adding more than 10 objects to any particular sector. If you wanted to limit the maximum number of textures to 32, then the world editor would prevent you from adding a 33rd texture. In other words, a custom world editor can offer arbitrary features or impose arbitrary restrictions to conform to the features or restrictions of the underlying 3D layer, so that the resulting world files are always valid.

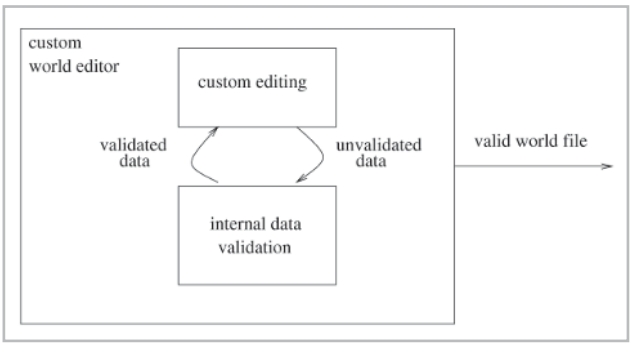


Figure 6-2: A custom world editor can be made to fit your 3D application exactly, only allowing valid world files to be created.

The flexibility of such a custom world editor comes at a high cost for programming the editor itself. As we saw in Chapter 3, a 3D editor can encompass many features; it can be a real challenge to implement these features in a reliable program which in the end truly saves you time. There are also a large number of basic operations which have to be implemented for a useful world editor, all of which take time to implement: object selection, multiple viewpoints, vertex selection,

transformations, and so forth. If you've ever written an editor of any kind (a music editor such as ModEdit, or a text editor, for instance), then you are surely familiar with all of the issues required to create a robust, reliable, efficient editing system. This is not meant to discourage you at all costs from writing your own level editor. By judiciously choosing a feature set and by reliably implementing a stable architecture, it is entirely possible to create a good level editing system which exactly matches your 3D application. Depending on the overall goals and time frame for your project, sometimes writing your own level editor can be the best choice.

Nonetheless, it can be frustrating to spend several weeks or months writing (and above all debugging) your own level editor when what you really want to be doing is creating an interesting, interactive 3D world. In such a case, we can turn to a third alternative: adapting an existing editor.

Adapt an Existing Editor

Adapting an existing editor means that we should find a 3D editor that has a feature set similar to the features we require for editing worlds. We then edit our worlds in the 3D editor, using the editor's usual tools for placing, scaling, orienting, or otherwise changing 3D objects. Plug-ins or external programs can be used to allow for customized editing of objects or their attributes in a manner needed by the underlying 3D application. The 3D editor should allow exporting the world to a documented output format which can then be parsed and transformed by an external program to create the final world file.

Taking this route requires you to understand the capabilities of the 3D editor, the needs of the world file, and how to write programs to transform between the two. Not all 3D editors are suitable for world editing. For instance, some 3D modeling programs focus primarily on the editing of just one object at a time. Such a 3D editor would be difficult to adapt for world editing purposes, where an important fundamental idea is the existence of multiple objects located in different sectors or regions of space.

Furthermore, if you use an existing 3D editor, it is likely that you will not be able to check for validity of the world files until after exporting the file from the editor. This means that within the 3D editor itself, you might be able to create worlds that the underlying 3D application cannot handle (for instance, if you exceed the maximum number of objects per sector or the maximum number of textures, or violate some invariant condition of the 3D application, such as non-overlapping rooms). An existing 3D editor typically cannot enforce application-specific constraints. Of course, you can build in an error-checking phase after exporting, to check the exported world file for validity, but this is slightly more inconvenient than having immediate error checking within the editor itself. This is the fundamental trade-off that has to be made: if you want exact control over all editing parameters within the editor itself, you probably have to write your own editor; but if you can adapt an existing system well enough and live with some minor inconveniences, you are probably better off using an existing editor.

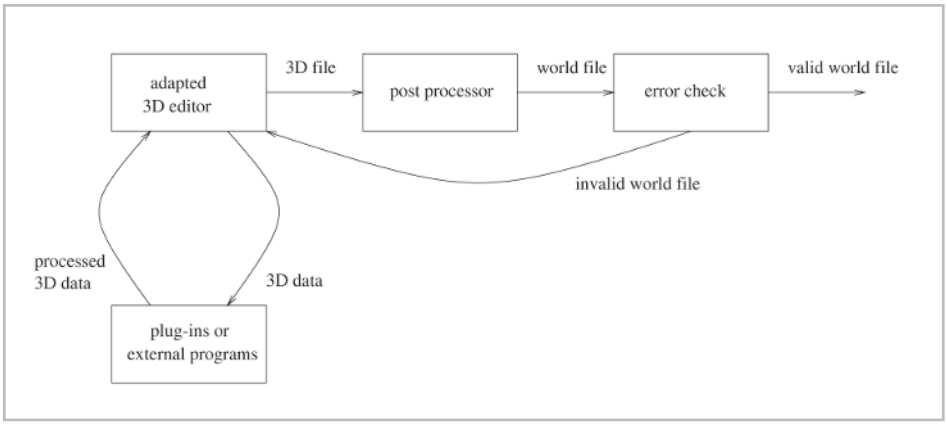


Figure 6-3: Adapting an existing 3D editor allows you to leverage existing technology, but means that custom editing must be done via plug-ins or external programs, and invalid world files are sometimes only detected after export from the editor.

The route chosen in this chapter is that of adapting Blender for world editing purposes. Let's now take a closer look at the exact requirements of this system, then proceed to see how the system works in practice.



NOTE Again, don't forget that the Blender-specific, portal-based world editing scheme presented here is just one of countless possible world editing schemes. At the end of this chapter, we look at some other ways of using Blender or another 3D editor for creating other types of worlds.

Using Blender for Portal Worlds

Since Chapter 3 used Blender to create 3D models for import into our l3d programs, it is only logical also to use Blender to create entire 3D worlds. The worlds we wish to create are portal worlds, defined by the world file format of Chapter 5.



NOTE Sometimes it makes more sense to separate model creation from level creation, as is done in the World Foundry system (see Chapter 9). One 3D package might be more suited to creating textured IK animated characters, while another 3D package might have better tools for placing rooms and objects and editing their properties.

Stated simply, the goal for our world editing system is to create valid world files. The world file definition implicitly defines the requirements on the world editing system. But let's state the requirements more concretely. We want to create portal worlds, which consist of sectors and portals which link the sectors. We also want to be able to place plug-in objects within the sectors. This means that our world editing system must allow us to do all of the following:

- Create separate areas of the world, which are the sectors in a portal/sector scheme
- Define the geometry and texturing of the polygons within a sector

- Define polygons as portals
- Link portals with sectors
- Place plug-in objects with sectors
- Define the geometry, texturing, position, orientation, and behavior of plug-in objects

These requirements are fairly modest, but given the fact that plug-in objects can have any appearance and behavior, defined via a plug-in file which is external to the world editing system, the system does allow for creation of interesting, dynamic worlds.

The next three sections look at one system for using Blender to reach these goals. We look at this world editing system from three different perspectives. First, we look at the Blender features we use, the limitations of those features, and ways of working around these limitations. Second, we look at the system from a step-by-step perspective, as you would view the procedure when creating a world. Finally, we look at the system in terms of data flow. After understanding the system from these three perspectives, we then proceed to create a simple world to see exactly how to use Blender and the external tools.

TEAM-FLY

Main Ideas of a Blender Portal World Editor

Let's now look at the main ideas of how we are going to use Blender in combination with some custom tools to fulfill the above requirements. The main ideas of our Blender portal world editing system are as follows.

- Meshes as sectors. We represent a sector as a single polygonal mesh object in Blender. The geometry of the mesh defines the geometry of the sector. A sector cannot be split across two meshes. Similarly, a single mesh should not contain the geometry for more than one sector. For a convex cell portal engine the geometry in each cell should be convex; for an arbitrary cell portal engine, using an additional means of VSD, the geometry in each cell need not be convex. Checking for sector convexity is the responsibility of the person doing the world editing.



NOTE A custom world editor could prevent the creation of any non-convex mesh. This again illustrates the fundamental difference between the choices of writing your own editor or adapting an existing editor. Adapting an existing editor allows you to leverage the existing editing framework, but also means that the creation of certain illegal features, such as non-convex sectors, is allowed within the editor and can only be detected later on.

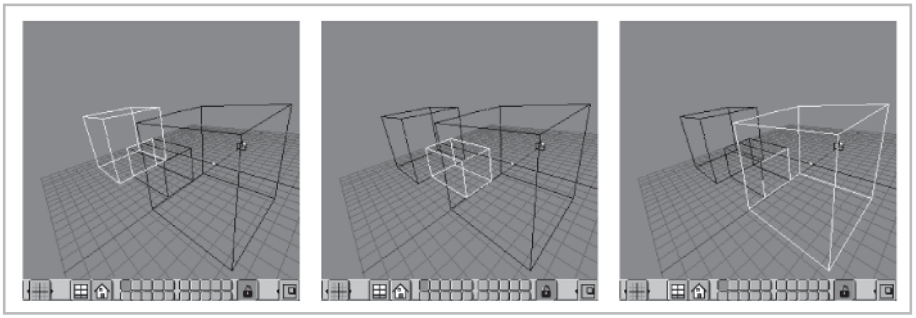


Figure 6-4: Each sector in a world is modeled as a separate mesh in Blender.

Team-Fly®

- Simple meshes as plug-in objects. We represent a plug-in object as a single polygonal mesh object. The actual geometry of the mesh is irrelevant; thus, we can use a cube or other simple mesh to represent the plug-in object. The cube's position and orientation determine the position and orientation of the plug-in object. Furthermore, we associate a plug-in name and a mesh name with the cube, which determine the behavior and appearance of the plug-in object at run time.

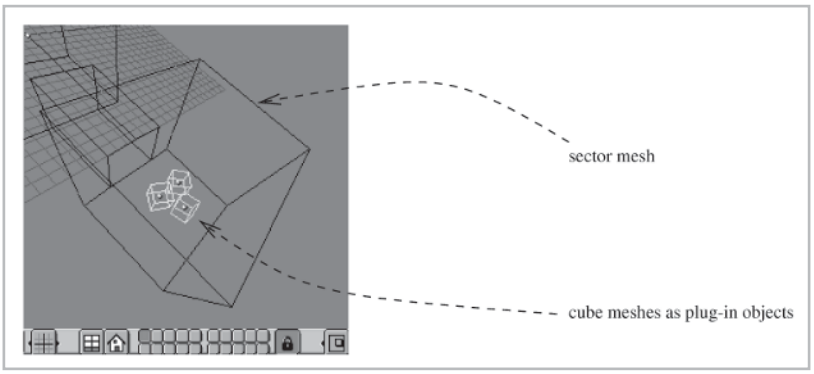


Figure 6-5: We use a simple placeholder mesh (such as a cube) to denote a plug-in object, whose geometry and behavior are determined at run time. We associate a plug-in name with the cube, via the external tool `blend_at`.

- Implicit portals. Portals are specified implicitly, not explicitly. Each sector must be a closed convex set of geometry; any holes in this geometry are then portals. (We define exactly how to find such holes later.) The portal's connected sector is the sector containing a portal whose coordinates correspond to the original portal's coordinates.

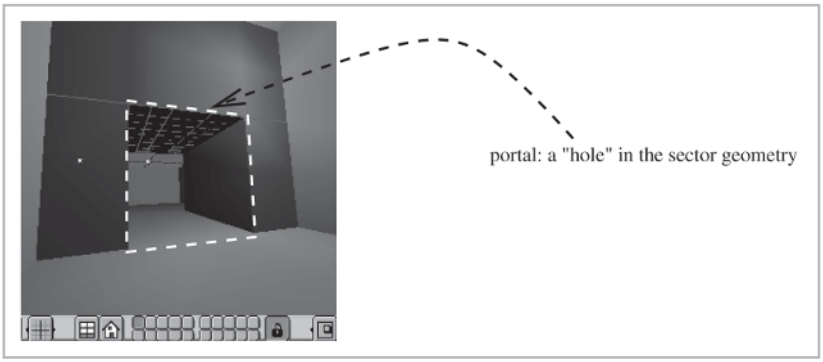


Figure 6-6: Holes in a sector's geometry are implicitly portals.

- Export to multiple Videoscape files. By marking all meshes (press **a**) and exporting them as Videoscape (press **Alt+w**), we can save all meshes to separate, sequentially numbered Videoscape files. This allows us to save an entire world, albeit as separate files. The separateness of the files creates a serious problem: the loss of name information.

- Loss of name information upon export. Although Blender's Videoscape export facility allows us to export multiple objects (i.e., an entire world) at once, there is major limitation: all exported objects have essentially random filenames. This means that after Videoscape export, we have a series of files which together make up the entire world; however, we do not know which specific file is which specific object or sector in the world. Being able to uniquely reference each object is important for any object-specific information (such as, this sector uses texture file `wall.ppm`) or for inter-object references (such as, object TORCH01 is initially located in sector ROOM02). Thus, we must have some way of uniquely referencing objects and sectors within the world before and after export, leading to the next point.

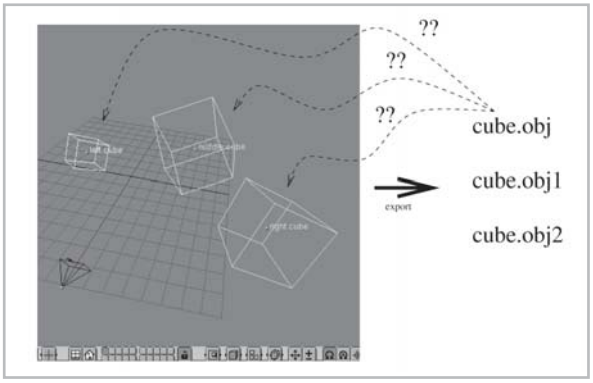


Figure 6-7: Loss of name information after Videoscape export. After exporting, we have three randomly named files. But which one is the left cube, which one is the middle cube, and which one is the right cube? With raw Videoscape files, there is no way to tell.

- Embedding identifiers within mesh geometry. After Videoscape export, all name information is lost for all meshes; only the mesh geometry remains. But by artificially adding overlapping vertices in a special way into the mesh geometry, we can encode a numerical identifier into the mesh itself, which remains within the mesh and is therefore preserved even after Videoscape export. In other words, we can use the mesh geometry itself to store a unique number within each mesh, thereby allowing unique identification of each mesh after Videoscape export. A separate tool is responsible for creating and later extracting such mesh identifier information.

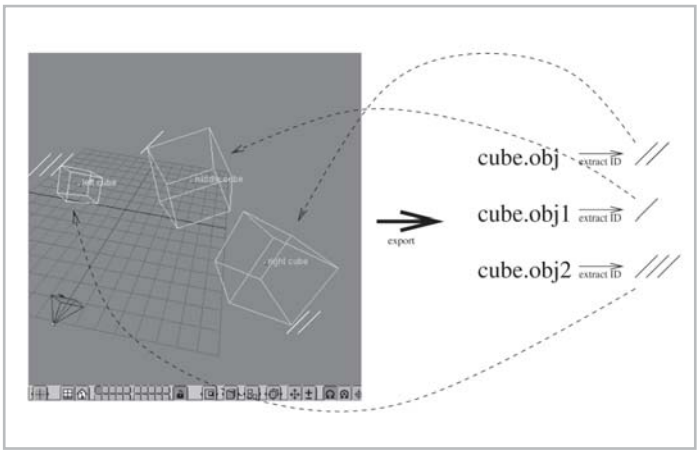


Figure 6-8: When storing redundant edges within the mesh geometry itself, these edges can be detected after Videoscape export to identify the mesh. Here, the redundant edges are represented schematically as diagonal lines stored within the mesh. After export, for a particular mesh file, we extract the diagonal edges again. The count of diagonal edges forms a unique numerical identifier allowing us to know which object is saved in the particular mesh file. Note that the diagram is only schematic; the actual geometric storage of the identifier, covered later, is more complicated and is stored as a series of physically overlapping edges.

- Associating arbitrary data with a mesh identifier. By tagging each mesh with a unique identifier, we can then create a separate database associating each mesh identifier with any arbitrary additional information we wish to store with the mesh. We could store the object's type (if it is a sector or a plug-in object), its name, any texture files used, its plug-in name if it is a plug-in object, and so forth. This additional information is then used when creating the final world file.

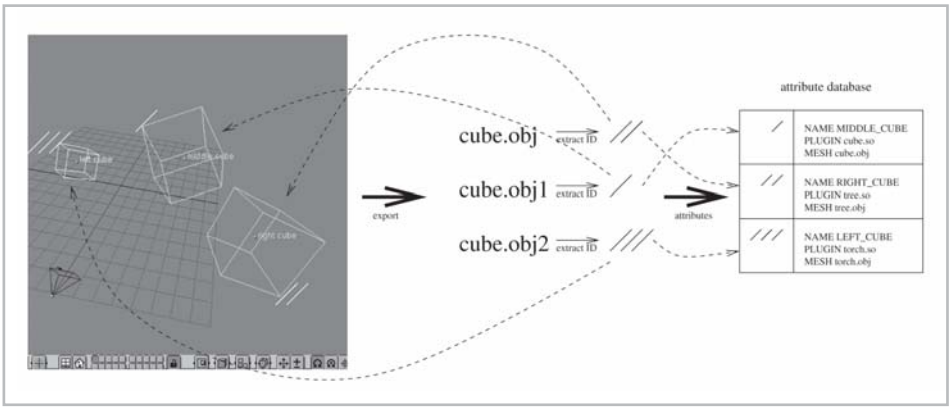


Figure 6-9: By storing a separate database of information associating each mesh identifier with an arbitrary list of attributes, we essentially are associating arbitrary attributes with each mesh object.

Step-by-Step Guide to World Design

We've now seen what features we use in Blender to create our portal worlds (meshes as sectors, implicit portals, embedding IDs into mesh geometry, using an external database to associate attributes with IDs). Let's now look at the system from the viewpoint of the world designer, i.e., the person actually using the system to create a world. From the viewpoint of the world designer, the entire system can be seen as the following step-by-step process.

1. Create sectors as separate meshes in Blender. The polygons within the mesh define the shape of the sector.
2. Tag each sector with an ID, using external tool `blend_at`.
3. Associate additional information with the sector ID, using external tool `blend_at`. Such additional information for sectors includes the type name (`SECTOR`), the name of the object, texture space definitions, and texture filenames used to texture the polygons of the sector.
4. Create plug-in objects as cubes in Blender. The polygons within the cube have no bearing on the final shape of the object at run time; the shape is determined by associating a separate mesh file with the object (see below). Position the cubes at the desired locations where the plug-in object should initially be created.
5. Tag each cube with an ID, using external tool `blend_at`.

6. Associate additional information with the object ID, using external tool `blend_at`. Such additional information for plug-in objects includes the type name (`ACTOR`), the name of the object, the plug-in filename, the name of the sector containing the object, the object's position and orientation, the mesh file defining the object's geometry, the texture image file, and the texture coordinate file.
7. Select all meshes (sectors and plug-in objects), and export them all at once as Videoscape files.
8. Use external tool `vid2port.pl` to create the world file from the exported meshes. The tool extracts the ID from each mesh, then finds any additional attributes associated with the ID in the attribute database. Next, the tool finds portals by looking for holes in the geometry of all sectors, and links portals to sectors by looking for overlapping portals in other sectors. Finally, given the mesh geometry and any additional attributes, the tool writes the appropriate lines in the world file, either `SECTOR` or `ACTOR` blocks.

Data Flow within the World Editing System

Figure 6-10 shows the flow of data which takes place when creating a world file from a series of Blender meshes. The data flow diagram does not show control flow; it makes no statement about any execution order. The purpose of the diagram is to show how Blender meshes get transformed into the world file.

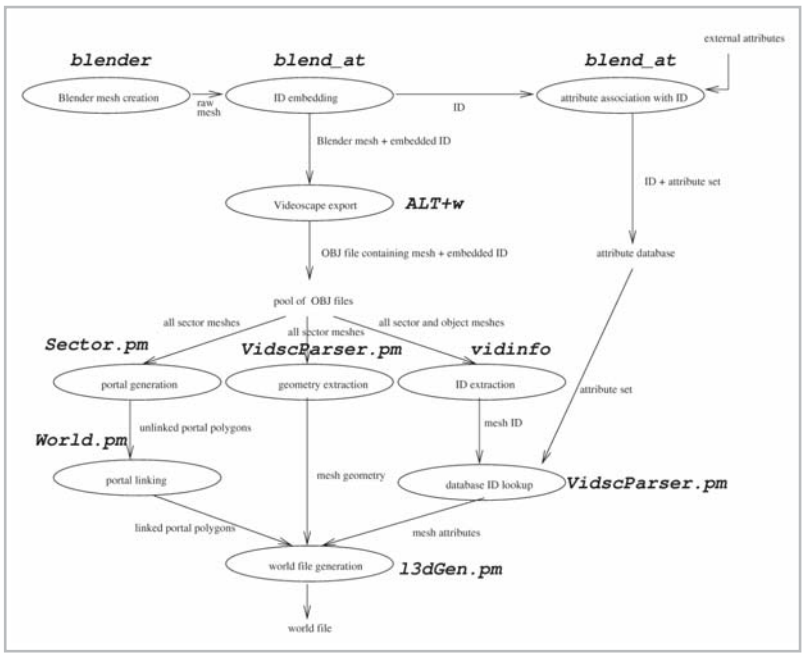


Figure 6-10: Data flow for creating a world file from Blender meshes.

Each arrow in the diagram represents a data flow, and is always labeled with the type of data flowing. Think of each arrow as a hose on an automobile engine; various gases must flow from part to part for proper operation, and the contents of each hose must always be known to

understand the system. Each oval in the diagram represents a process that takes some data as input, and produces some other data as output. Next to each oval, in a bold italicized font, is the name of the actual program which is responsible for that process. In the course of this chapter, we discuss each of these helper programs individually. Items not contained within an oval (the attributes database, the world file, the external attributes, and the pool of OBJ files) are external data sources or sinks which do not process, but merely provide or store data.

This diagram has a clearly defined semantic interpretation. This means that there is a set of rules making it possible to mechanically go from the diagram to an English sentence which states something useful about the problem. (There is nothing more confusing than an important diagram with ambiguous semantics, where symbols implicitly represent several ideas. For instance, those with no formal training in software often indiscriminately use lines within a single diagram to simultaneously represent hierarchy, aggregation, control flow, data flow, and temporal dependency. Extracting precise information from such a diagram can be very difficult.) Here, the semantics are simple. For each oval, begin a sentence with "The process of" followed by the process name within the oval. For each arrow coming into an oval, substitute the phrase "takes as input" followed by the label of the arrow. For each arrow leaving an oval, substitute the phrase "produces as output" followed by the label of the arrow.

For instance, we can read the first few parts of the diagram as follows. The process of Blender mesh creation produces as output a raw mesh. The process of ID embedding takes as input a raw mesh, and produces as output a mesh with an embedded ID, and an ID. The process of attribute association with an ID takes as input an ID and external attributes, and produces as output an ID+attribute set. You can read the rest of the diagram in a similar fashion.

Now, having looked at the Blender portal editing system from three separate perspectives, let's see how the system works in practice, by creating a simple world in Blender and exporting it. This first example deals only with sectors and portals; the next example deals with textures and plug-in objects.

Creating Sectors and Portals

Sectors and portals in the world editing system are closely related. As we have said, a portal is not explicitly defined; instead, it is a hole in the geometry of a sector. Let's formalize this notion of a hole in a sector. A hole in a sector is a continuous loop of *free edges*. A free edge is an edge in a sector which is used by only one polygon within the sector. Such free edges have a polygon on just one side, and empty space—in other words, a hole—on the other side. This definition assumes that the sector geometry is convex and closed; in other words, you should not have free edges or holes which do not belong to portals.

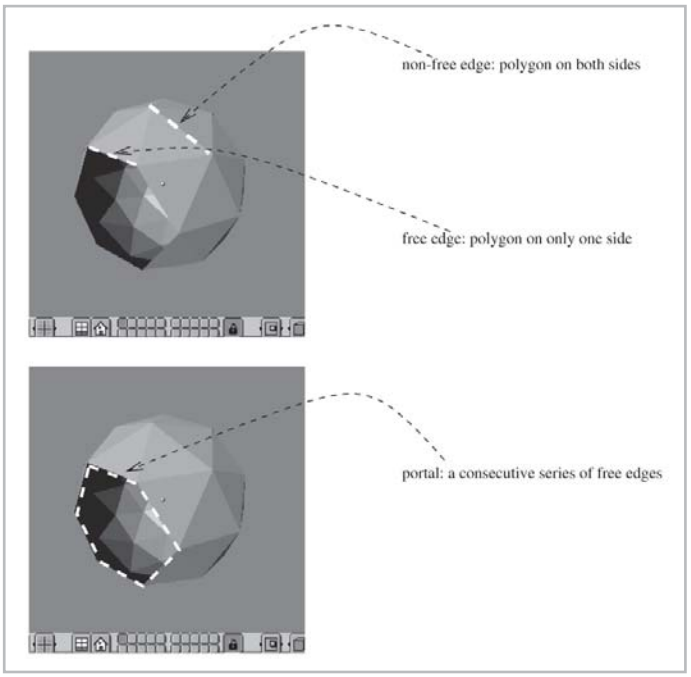


Figure 6-11: Free edges have a polygon on just one side. A consecutive series of free edges forms a hole in the sector, and thus is a portal.

Note that such an implicit definition of a portal rules out the possibility of two exactly adjacent portals sharing one edge (such a portal would appear to the viewer to be one portal), since the shared edge in this case has no geometric polygons on either side, just portal polygons, meaning that the edge is not saved by Blender as part of the geometry. Even if such an invisible edge between adjacent portals were saved with the geometry, we still would not know whether one large portal or two smaller ones were wanted, since both would form valid loops of free edges. With the system described here, specifying adjacent portals is not possible; because of the lack of a separating edge, the two adjacent portals are automatically combined into one larger portal. This would be incorrect if we wanted two adjacent portals which lead to different sectors. Supporting such cases would require explicit rather than implicit specification of portals, and would probably require writing a custom world editor from scratch. The reason for this is that polygons in Blender can only have three or four sides, but portal polygons can have a greater number of sides. Since Blender doesn't allow us to specify arbitrary convex polygons with more than four sides, this means that explicit portal specification, and the ability to have two exactly adjacent portals leading to different sectors, requires another tool. Note that the l3d classes for handling portals have absolutely no problem with adjacent portals; the only problem here is the specification of such adjacent portals within a modeling tool.

Portal connectivity (which sector is connected to a portal) is also implicit. Once we have a list of portals (holes) in each sector, we then search for portals whose vertices exactly overlap. Each such portal is then connected with the sector of the other portal.

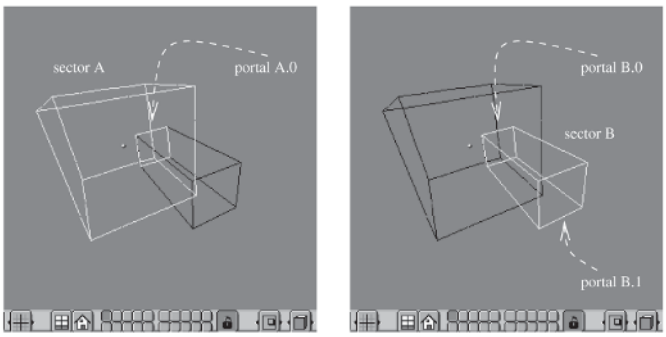


Figure 6-12: Portal connectivity is implicitly determined by overlapping of portals. Here, portal A.0 of sector A exactly overlaps with portal B.0 of sector B. Therefore, portal A.0 is connected with sector B. Also, portal B.0 of sector B exactly overlaps with portal A.0 of sector A; therefore, portal B.0 is connected with sector A. Portal B.1 in this figure is not connected to any sector.

This implies that precise alignment of the vertices of the portals in adjoining sectors is essential to correct operation of this system. If portals from adjacent sectors do not align exactly, then the world editing system will not be able to find overlapping portals, and will thus be unable to connect portals to their target sectors.

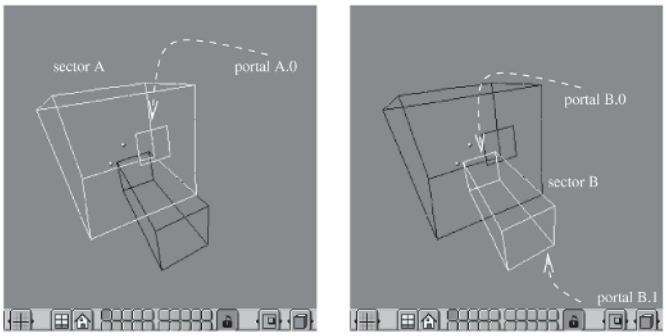


Figure 6-13: Slightly misaligned portals means that no corresponding overlapping portals will be found, and that portals are left unlinked. Here, no portals overlap with one another, meaning that no portal connectivity could be automatically determined.

Let's now see how we can best create such sectors and portals within Blender, so that the portals line up.

Tutorial: Creating Aligned Portals via Extrusion and Separation

The best way of aligning the vertices of portals is to initially create separate sectors as one mesh, then later separate parts of the mesh into individual sectors. The idea is to cut a portal into a sector by deleting some faces. Then, we select the vertices along the edges of the deleted faces and extrude these vertices, producing a tunnel which exactly lines up with the portal we created. Finally, we separate the original sector and the extruded tunnel into two separate meshes, automatically preserving the overlapping of the portals. The following tutorial illustrates how to do this.

1. Begin with an empty Blender window. Ensure that Draw Faces is activated in the EditButtons (**F9**).
2. Create a simple cubical sector. Press **Shift+a**, select **Mesh**, then **Cube**.

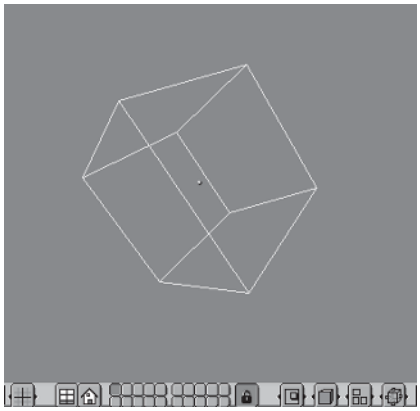


Figure 6-14

3. Select the vertices of a face in which you wish to have a passageway to another sector. You can either use border select mode (press **b**) or you can individually **Shift+right-click** each of the desired vertices.
4. Subdivide the face until enough smaller faces appear so that you can remove a few to create a small hole in the face. With a face selected, subdivide it as follows: press **w**, then select **Subdivide** from the menu. Here, we have subdivided the far face twice, yielding 16 smaller faces.

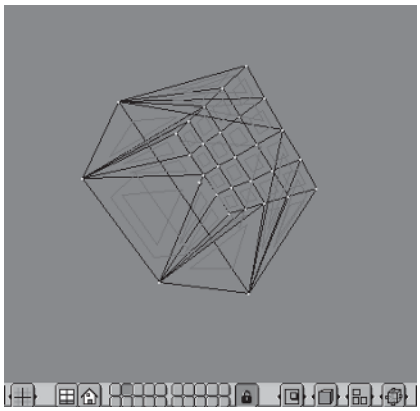


Figure 6-15

5. Select the vertices of the faces which you wish to remove, and delete the faces only by pressing **x**, then select **Faces**. The faces should all lie within one plane. Here, we have deleted the four faces in the bottom center of the subdivided face, creating a small doorway.

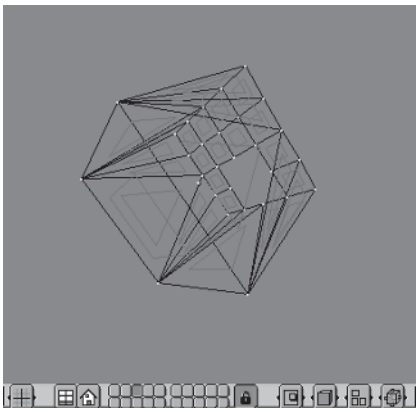


Figure 6-16

6. Select the vertices lying on the edges of the faces you just deleted. Extrude the vertices by pressing **e**, **Enter**, drag the mouse, and left-click to finish. It is easiest to perform the extrusion in the top, front, or side orthogonal views (press **7**, **1**, or **3** on the numeric keypad to choose these viewpoints, and press **5** to toggle between orthogonal and perspective viewing). This extrusion creates a tunnel or corridor.

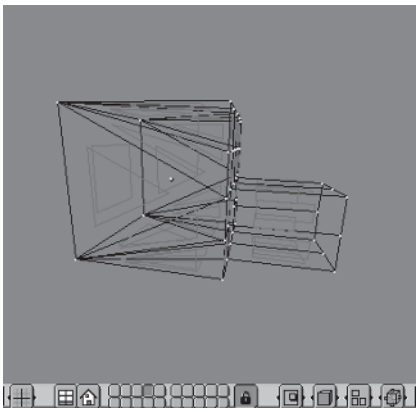


Figure 6-17

7. Select just the vertices of the original mesh, and separate them from the newly extruded tunnel by pressing **p**, **Enter**. The selected vertices are then removed from the current mesh and moved into a new, separate mesh. The vertices of the open part of the tunnel exactly match up with the vertices in the open part of the original sector mesh, meaning that the two portals between the two sectors exactly overlap.

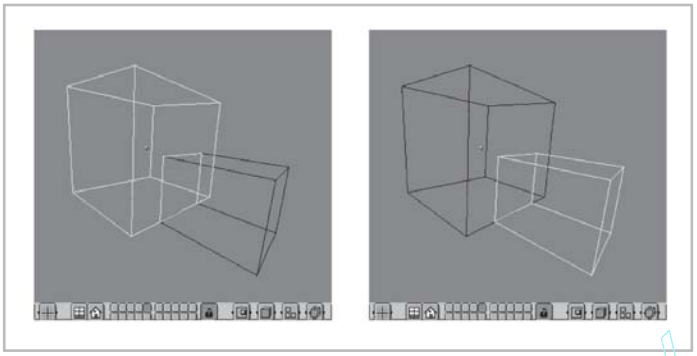


Figure 6-18

By repeatedly extruding sectors in this way, we cause the creation of corridors or rooms that are built upon the existing vertices of the polygonal portal. Then, by separating the geometry into separate meshes, we cause exact duplication of the portal vertices. Thus, the general procedure is: create a mesh, cut a hole to form a portal, extrude the portal, separate the mesh into two sectors.

Tutorial: Aligning Portals from Separate Meshes

The previous example showed you how to create a mesh via extrusion of a portal, and separation of the mesh into pieces. But sometimes you already have two separate sectors, which were created independently and not via extrusion of a portal. You can also connect these sectors via portals. To do this, each sector must have a portal with the same number of vertices and edges. The idea is to join the meshes together into one mesh, manually eliminate duplicate vertices around the portals so that both meshes share exactly the same vertices along the portal, and then separate the meshes again so that the resulting portals align exactly.

The following tutorial shows you how to align portals from two separate meshes. The files from this example may be found in directory `$L3D/data/levels/tworooms`.

1. Create a cube with an extruded tunnel as in the last example.

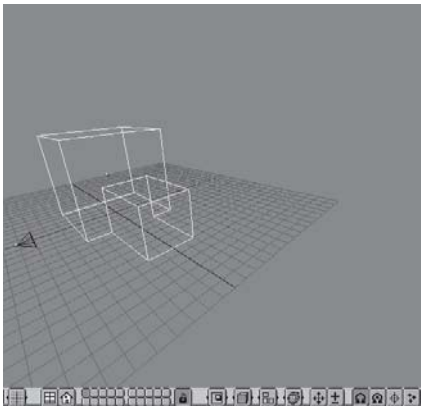


Figure 6-19

2. Exit EditMode. Duplicate the mesh so that there are two copies of the mesh: select the mesh, press **Shift+d**, move the copied mesh to its new location, and left-click.

Team-Fly®

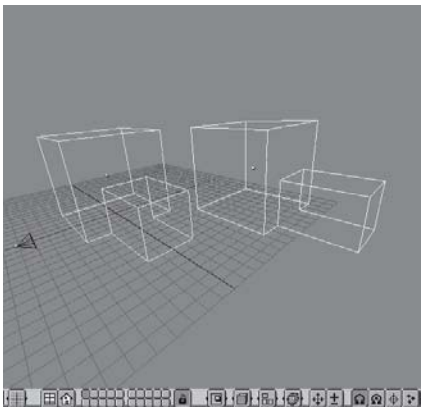


Figure 6-20

3. To make this example more difficult—and more realistic—switch to top view and rotate the meshes so they are positioned diagonally facing one another. This makes the example more difficult because the portals now lie on an arbitrary plane in space instead of being neatly axis-aligned. When working with real models, you will encounter arbitrarily aligned portals.

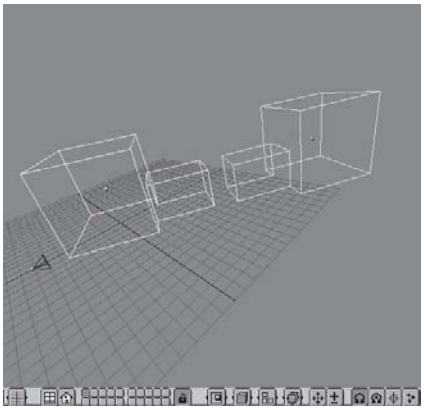


Figure 6-21

Now, we have a typical situation where we have two separate, arbitrarily aligned meshes, each having a portal with identical vertex and edge counts. What we now want to do is join the two tunnels with one another via a portal.

1. Move the meshes together so that the portals align as closely as possible.

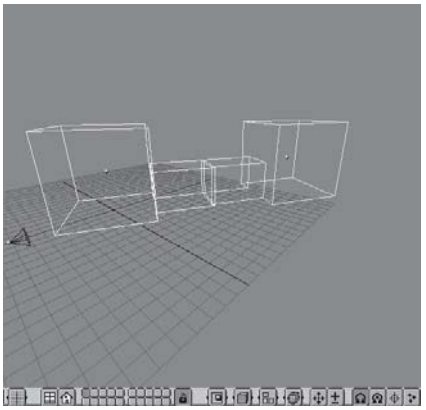


Figure 6-22

2. Select both meshes and press **Ctrl+j, Enter** to join the selected meshes.
3. Enter EditMode by pressing **Tab**. Select all vertices except for those around the portals. Hide all of these vertices by pressing **h**.

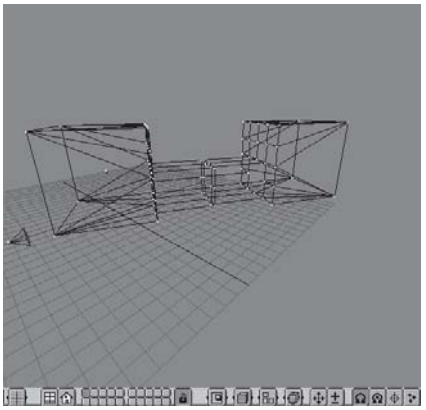


Figure 6-23

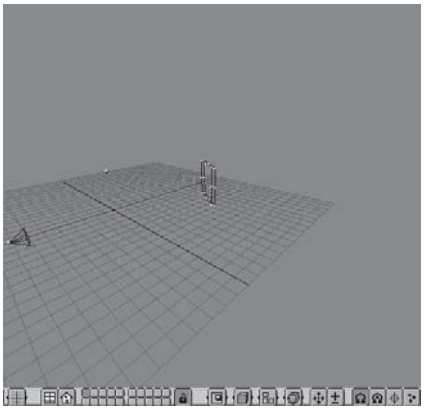


Figure 6-24

4. Move each vertex of each portal as close as possible to its corresponding vertex in the other portal with which it should overlap. Move vertices as usual by right-clicking and pressing **g** to grab them, then using either the arrow keys or the mouse to move the vertex to the desired position and pressing **Enter**. Switch frequently among top, front, and side views, zoom in closely, and also use the perspective view mode with interactive view rotation to check if the vertices really overlap. The danger is that they might appear to overlap from one viewing angle, but not from another; thus the importance of frequently switching among viewpoints while editing and of interactively rotating the view in perspective.

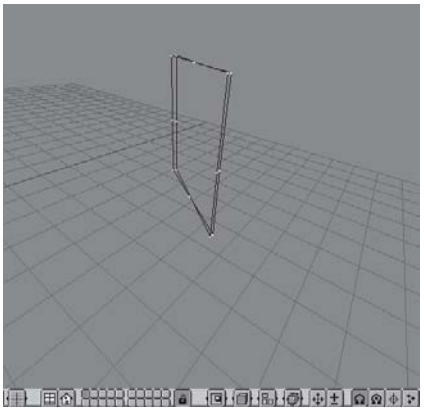


Figure 6-25

5. Select all portal vertices for the two portals which should overlap. Ensure that all vertices from both portals are selected; temporarily start then cancel a grab operation to verify this. If any vertices are not selected, they will appear to stick to their old positions during the temporary grab operation; in this case, use border select mode to select the still unselected portal vertices.
6. In the EditButtons, find the button Rem Doubles. This button removes overlapping or nearly overlapping vertices. Directly beneath this button is a NUMBUT labeled "Limit." This button controls the minimum distance below which two vertices will be merged into one. Depending on how close you were able to move the vertices to one another, you will need to change this limit. For this example, the limit was set to 0.35.
7. After setting the limit for double vertex removal, click **Rem Doubles**. Blender then displays a message showing how many doubles were removed. The count should be the same as the number of vertices in one of the portal polygons; if not, then the limit was set too low. In this case, set the limit slightly higher, and repeat step 6.

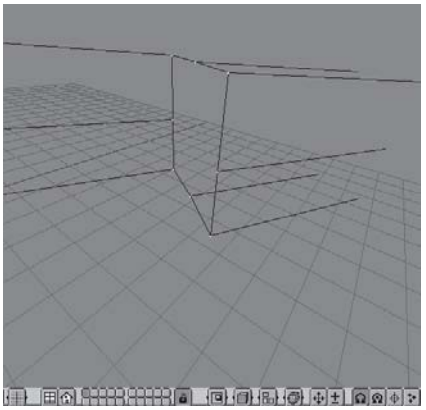


Figure 6-26

8. After you believe that all doubles have been removed, verify this as follows. Select each individual vertex of the portal. Press **g** to temporarily enter grab. Move the mouse slightly to see if another vertex is lying "underneath" the selected vertex. After looking underneath the vertex, press **Esc** to cancel the grab. If any such vertex appears underneath the currently selected one, then it is a double vertex which has not yet been removed; thus, increase the limit and repeat step 6. If no such vertices appear underneath any of the portal's vertices, then all doubles have been removed, and the same portal vertices are now shared among the two meshes.
9. After removing the doubles, it is likely that the vertices of the portal no longer lie within a plane. Ensuring that the vertices do lie within a plane is important since otherwise the portal will be non-planar, and portal crossing computations based on collision detection (see Chapter 8) will not function reliably. It is not absolutely essential to align the vertices in a plane (in other words, l3d programs will not crash if the vertices are not completely aligned), but it is best to try. Align the vertices in a plane as follows.
 9a. Select all vertices belonging to the portal.
 9b. Switch to orthogonal top view. Remember the approximate orientation of the portal.

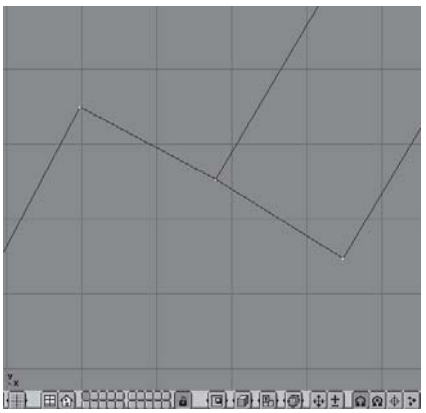


Figure 6-27

 9c. Rotate the portal so that it is parallel to a grid line. Press **r**, move the mouse to rotate, and press **Enter**.

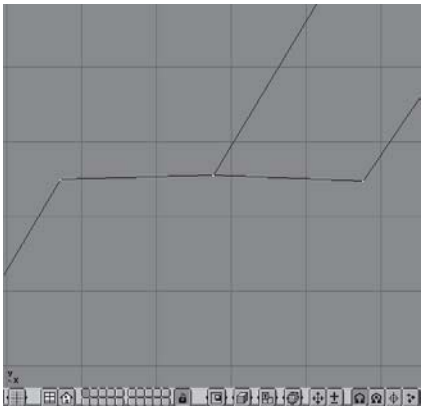


Figure 6-28

9d. If necessary, repeat the rotation process in orthogonal side view, again remembering the approximate original orientation in side view. Such a rotation would be necessary if the portal were tilted around the *x* axis so that it was not perfectly straight up and down. For this example, the portal is not tilted around the *x* axis, so no side view rotation is necessary. After such a side view rotation, switch back to top view. At this point, the portal should be aligned more or less perfectly vertically, and you should be looking straight down along the top edge of the portal.

9e. Adjust the grid size to be small enough so that the grid points (the intersections between the lines) are located fairly close to the portal vertices. In the 3DWindow, press **Shift+F7**, enter the desired grid spacing in the NUMBUT labeled "Grid," then press **Shift+F5** to change back to the 3DWindow. Here, we have reduced the grid spacing from 1.0 to 0.1.

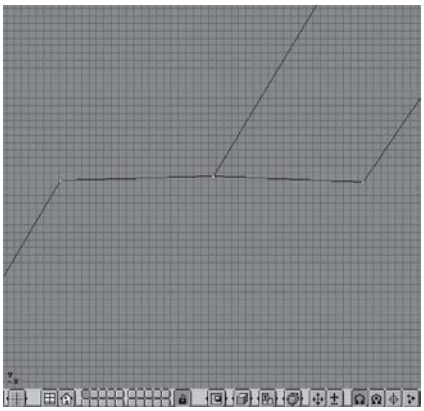


Figure 6-29

9f. Press **Shift+s**, then select **Sel->Grid** from the menu. This snaps all selected vertices, which are all portal vertices, to the nearest grid point. Now it is clear why we reduced the grid spacing in the previous step: we don't want the portal vertices to be snapped to vertices very far away, which would distort the shape of the portal.

9g. Now, all portal vertices lie on grid lines but not necessarily on the same grid line. Manually select and grab portal vertices so that from top view, they all appear to lie on the same grid line. Press and hold **Ctrl** while grabbing the vertices to ensure that they remain exactly on the grid lines. At this point, the portal should be completely planar, lying within the plane of the single grid line containing the vertices. Rotate the view interactively in perspective mode to verify that it looks approximately planar.

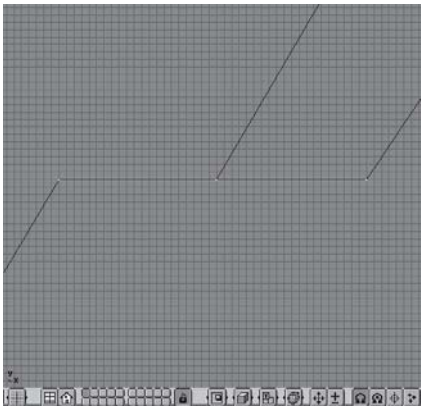


Figure 6-30

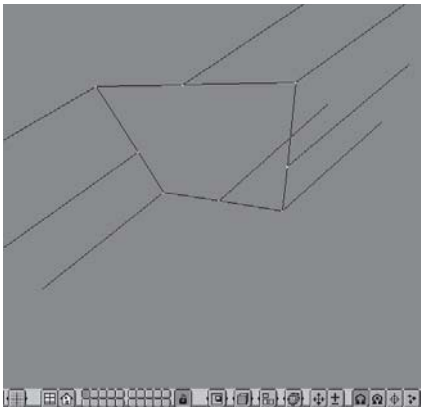


Figure 6-31

9h. Select all portal vertices, switch to orthogonal top view, and rotate the portal so that it has its original orientation as seen from top view. In other words, you now undo the rotation you did earlier. Do the same in orthogonal side view, if you rotated the portal in side view. The portal is now planar, but with its original orientation.

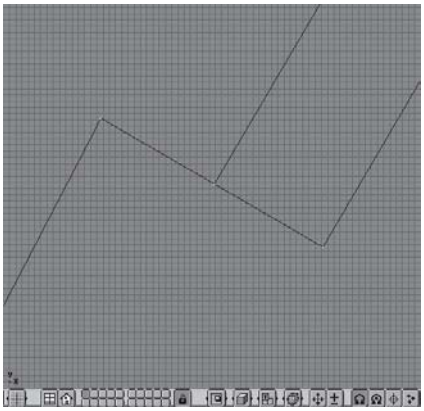


Figure 6-32

9i. Press **Alt+h**. This again displays all hidden vertices.

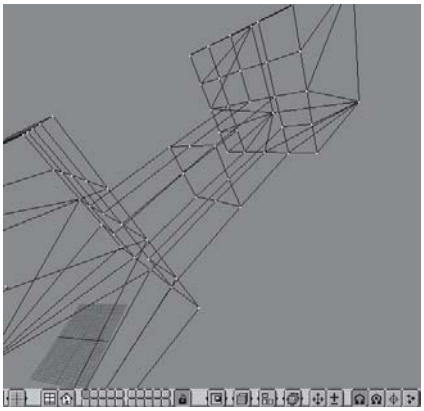


Figure 6-33

9j. Conceivably, through the process of snapping portal vertices to make the portal polygon planar, other four-sided polygons which shared the same vertices are no longer planar. (This is not an issue with triangles, since a triangle is always planar.) Visually check for any such anomalies, and make any such polygons planar by a similar process: select polygon, hide everything else, rotate, snap to grid, align to one single grid line, rotate back.

10. Finally, select the vertices of each convex part of the mesh, and press **p, Enter** to separate these into separate meshes. At this point, you have separate convex meshes, which line up exactly on the portals.

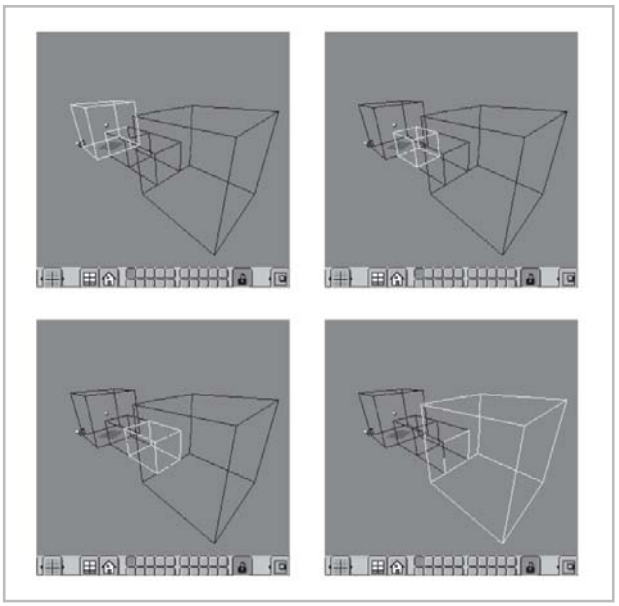


Figure 6-34

To summarize the idea of aligning portals, we join meshes together and eliminate duplicate vertices around the portal so that both meshes share identical vertices along the portal boundary. Then, we make the portal planar by rotating it into a convenient grid-aligned orientation, snapping it to a plane on the grid, then rotating it back. Finally, we separate the mesh back into individual pieces, which automatically causes duplication of the shared vertices along portal boundaries. This means that the portals align exactly.

Tips for Working with Portals

Now that we understand how to create sectors with portals that line up exactly, let's review some tips for effectively working with this system. We saw most of these tips in the tutorials above, but the last few tips in the list are new.

- Temporarily grab vertices to see if there are any hidden vertices underneath a given vertex; cancel the grab with Esc.
- Switch often between multiple viewpoints, and rotate the view interactively in perspective mode, to get a spatial feeling for which vertices are selected. Also, temporarily grab the selection and move the mouse to move the selection around; this way, you can verify which vertices are selected.
- Snap vertices to the grid to create precise alignments. For exact planar alignment on arbitrarily oriented portals or polygons, first rotate the polygon to align with the grid, snap the vertices to one grid line, then rotate the now-planar polygon back to its original position. Exact alignment of two vertices, as required for portal joining, can be achieved by manually moving vertices close then increasing the Rem Doubles limit, or by snapping vertices exactly to one another as follows: select the first vertex, press **Shift+s**, select **Curs->Sel**, select the second

vertex, press **Shift+s**, then select **Sel->Curs**. This snaps the cursor to the first vertex, then the second vertex to the cursor, effectively snapping the second vertex to the first vertex.

- Activate the DrawFaces button in the EditButtons. Sector geometry should be defined by faces; portals should be holes in the sector without faces.
- Subdivide a large face to create smaller polygons which can then be removed to create a small portal in the large face.
- Add extra faces to simplify portal outlines. In the example we have looked at so far, we took a cube, subdivided a face twice, removed four faces, then extruded along the hole we created. While this works, it creates more polygons than are necessary. Notice that there are eight vertices along the portal border because of the face subdivision. Since the portal is square and not octagonal, we actually only need four vertices, not eight, along the portal border.

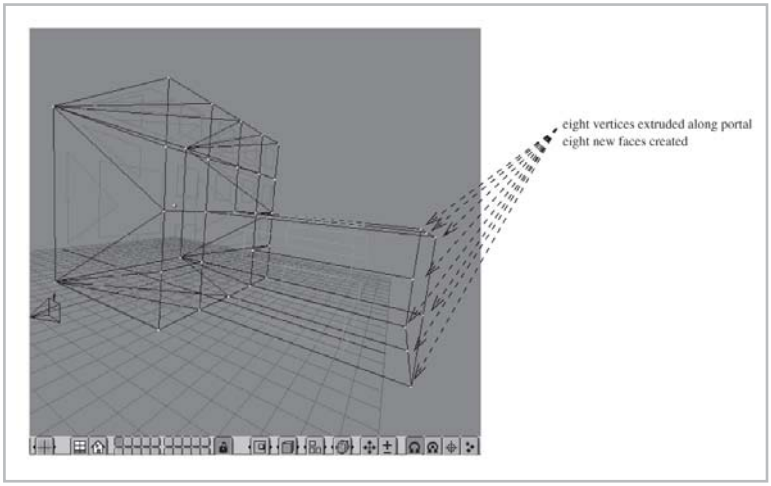


Figure 6-35: Eight vertices along the portal border. The edges all form a square, not an octagon, implying that four of the edges in this case are redundant.

By introducing extra faces along the edge of the portal, we can reduce the number of edges. We do this by inserting extra triangles along the portal border. For each set of two edges lying in a straight line, we push the meeting point of the edges up slightly, and introduce a new face formed by the triangular indentation thus created. (To create the new triangular face in Blender, select the three vertices after displacing the meeting point as described and type **f**.) In this way, the edge along the portal gets reduced to just one edge, instead of two.

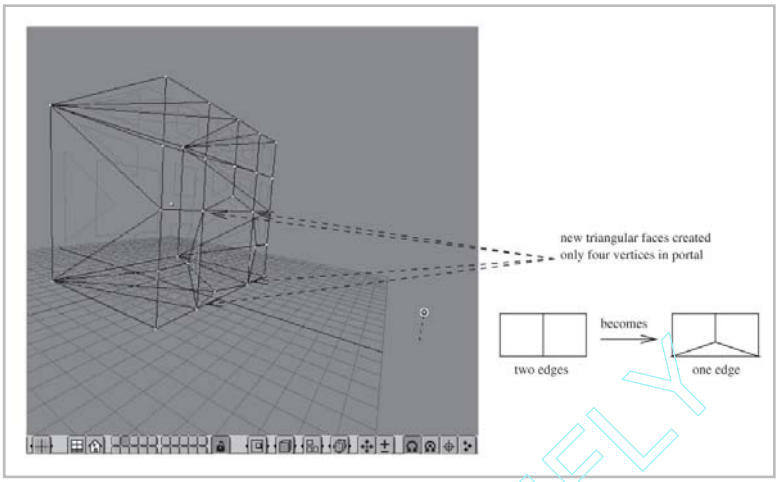


Figure 6-36: Introducing triangles along the portal border reduces the number of vertices in the portal.

Then, if we extrude the portal to create a tunnel, fewer polygons are created in the tunnel. This can add up to a significant savings if the tunnel is extruded several times to create several tunnel segments; each segment now has only four polygons, instead of eight as before.

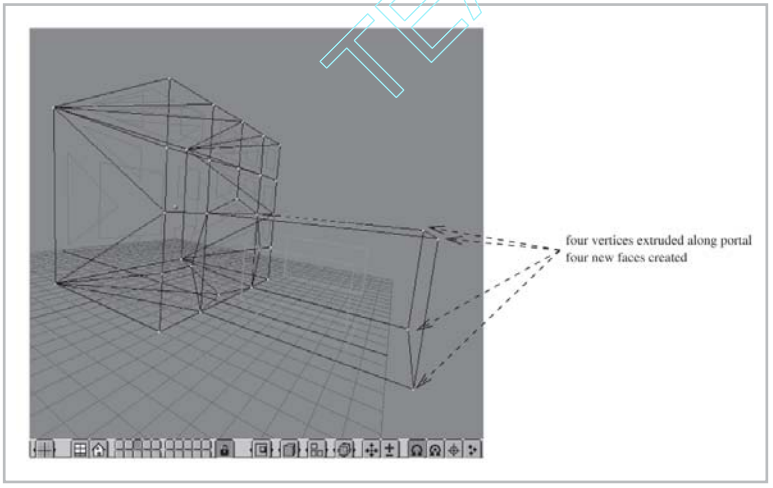


Figure 6-37: Extruding the reduced portal creates correspondingly fewer faces.

- Check normal vectors. It is important that the normal vectors of a model face in the proper direction, so that back-face culling works properly. If you get the normal vector orientation incorrect, the symptom at run time is that from the inside of the sector (which is the normal viewing situation), some polygons simply fail to appear, because they have been removed by back-face culling. With sectors, the normals should all point inward, since a sector is geometry which is viewed from the inside. As covered in the introductory companion volume *Linux 3D Graphics Programming*, you can check the normals by activating the TOGBUT Draw Normals in the EditButtons; flip the normals inside by selecting the desired faces and pressing **Ctrl+Shift+n**.

TEAMFLY



NOTE Note that on rare occasions Blender can get confused as to which side is the "inside" side of a mesh, so always visually check the normal vector orientation yourself, and flip the normal vectors for individual faces if their vectors happen to point outside the mesh. If Blender is confused about the inside/outside orientation, you would then press Ctrl+n to point the normal in the direction Blender considers to be "outside," since in such a case this is really the inside direction. This has only happened to me once or twice, but it can happen.

- Fly through the scene to interactively get a feeling of how the level will appear when imported into l3d. To fly through a scene in Blender, switch to camera view (press **0** on the numeric keypad), then press **Shift+f** to enter fly mode. Move the mouse toward an edge of the screen to point the camera in that direction. Left-click to increase velocity in the forward direction; middle-click to decrease velocity. Press **Esc** to terminate fly mode and return the camera to its original position; press **Space** to terminate fly mode and leave the camera in its current position. Note that flying straight up in fly mode causes confusing turbulence, presumably because Blender uses successive concatenations of separate *x*, *y*, and *z* axis rotation matrices instead of the rotation matrix about an arbitrary axis. By using a concatenation of three separate rotation matrices, a situation called *Gimbal lock* occurs, where freedom to move in one direction becomes restricted because one axis has been completely or almost completely rotated into another, thereby making the static concatenation order of the separate rotation matrices invalid for certain rotational combinations—such as flying straight up.

Portalization: Generating Portal Connectivity

Now, let's look at how to generate a simple world file based on the sector meshes we created in the last tutorial. The most significant thing that this process achieves is the automatic generation of the portal connectivity among the sectors and the creation of the appropriate `PORTALPOLY` lines in the world file. We call this process *portalization*. This is significant because this connectivity information was only implicitly specified in Blender through portal alignment; after executing the commands below, the connectivity information becomes explicit, in the form of `PORTALPOLY` specifications. The next most significant thing that this process achieves is the creation of the rest of the world file, defining the normal `GEOMPOLY` polygons for the sector. This is convenient, but not as significant as the portal generation, because the geometry was already explicitly specified in the exported Videoscape files; we simply need to slightly rewrite the geometry specification to conform to the world file format. Taken together, these two aspects (generation of portal connectivity and definition of sector geometry) allow us to generate a usable portal world file from the exported Videoscape files.

For now, you should just execute the commands presented below; afterwards, we look at the actual code behind these commands in detail. Also, we haven't yet covered creating plug-in objects or how to texture the polygons in the sector; these topics come later in this chapter. Right now, it's important to get a feeling for how the basic system works before diving into the specialized and rather arcane solutions needed for plug-ins and texture assignment.

Begin with the same Blender file from the last tutorial, which should now contain exactly four meshes (two larger rooms and two smaller corridors). Again, this file is also located on the CD in directory `$L3D/data/levels/tworooms/tworooms.blend`. Then, proceed as follows.

1. Type **a** until all meshes are selected.
2. Press **Alt+w** to write all meshes to sequentially numbered Videoscape files. Choose any directory in which to save the files; as a filename, enter `tworooms.obj`.
3. Enter the following command, replacing `<PATH>` with the directory name where you saved the Videoscape files from step 2. This command executes the Perl script (which we cover shortly) to create portals and sectors from the separate Videoscape mesh files we just saved. The program prints lots of diagnostic output (free edge scanning, finding overlapping portals) to the standard error stream, and finally prints the actual world file to the standard output stream. Here, we redirect the standard output into file `world.dat`.

   ```
   perl $L3D/source/utils/portalize/vid2port.pl <PATH>/two-rooms.obj* > world.dat
   ```

 After executing this file, we have a mostly correct world file. We now must make a few manual changes to the world file. Later, these changes will be made automatically by the tool `blend_at`, but for now, we must do it by hand.
4. Edit the world file. Change the first line from "0 TEXTURES" to "1 TEXTURES". Immediately after this line, insert a single line containing the string "stone.ppm." By doing this, we add one texture file, `stone.ppm`, to the world file. We must add at least one texture file because by default, if we don't specify otherwise, all polygons use the first texture in the world file, with a standard mapping (texture space axes aligned with the polygon's edges). Later, we see how to specify custom texture files and mappings for sector geometry, in which case the specified texture files automatically get added to the world file.
5. Change the line "4 SECTORS AND ACTORS" to "5 SECTORS_AND_ACTORS." We are going to add one actor, the camera, to the file. Later, we see how to use tool `blend_at` to specify the camera location within Blender itself.
6. At the end of the file, insert the following line to specify the camera location.

   ```
   CAMERA CAM1 noplugin.so 0 0 0 1 0 0 0 1 0 0 0 1
   ```

 The first three tokens are the type of object, the name of the object, and the plug-in name (currently ignored by the l3d code for the camera object, though the code could be extended to allow for camera plug-ins). The next three numbers are the location of the camera; the last nine numbers are the orientation of the camera (currently ignored by l3d).

At this point, the world file should be usable. Copy it into the same directory containing the executable file `porlotex` from the last chapter. Also ensure that the referenced texture file, `stone.ppm`, is in the same directory. Then, execute the `porlotex` program as before; the program loads the new `world.dat` file and displays the world you just created in Blender.



NOTE Currently the world class does no error checking during the loading of the world file. Thus, any errors will likely lead to a program crash. Be sure to follow the above instructions carefully. With your own world files, if the program crashes during or shortly after loading the world file, trace the code with the debugger to find the exact problem.

After starting the program with the new `world.dat` file, you will probably first see a black screen. This is because we specified the camera position to be (0,0,0), but we inserted the camera into the last sector in the file (since the `CAMERA` line in the world file always places the camera in the immediately preceding sector in the world file). The last sector in the world file can be any one of the meshes you created in Blender; due to the loss of name information on Videoscape export, we don't know which of the original sector contains the camera; we just know that it is the last one in the world file. Thus, most likely, the camera will initially be located outside of the sector to which it is assigned. So, press **j** and **l** a few times to rotate the view and find the sector geometry, then navigate inside the sector. At this point, you can explore the world you just exported from Blender. Notice that the default texture assignment looks somewhat odd; we see later how to control the texture assignment exactly.

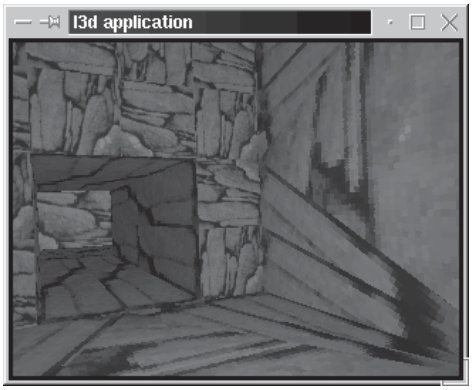


Figure 6-38: Output from program `porlotex` (from Chapter 5) with the newly created `world.dat` file. (A lamp has been placed in the scene by pressing p.)

Now, let's look a bit more closely at the actual code which converts from the Videoscape files into the portal world file.

Perl Scripts

The conversion from the Videoscape files to the portal world file is done via a series of scripts written in the Perl language. Let's briefly discuss what Perl is, and why we use it for the conversion work.

Perl is the Practical Extraction and Report Language. It is a high-level language that is very good at scanning text files, extracting information from these files, and printing reports based on these files. What this means in practice is that tasks which require you to parse and extract information from large amounts of text are very well suited for Perl. The Videoscape file format is text, and lends itself well to Perl processing.

What makes Perl particularly good for working with text is the integration of several typical Unix text processing commands and ideas into the Perl language itself. You can treat the input text as one large list of lines, search through it with a single `grep` command, sort it with `sort`, print it, and so forth. Perl is also a dynamically typed language, making it easy to quickly build and test systems. Perl's basic data structures include scalars (such as strings or numbers), arrays of scalars indexed by number, and associative arrays of scalars (known as *hashes*) which are indexed by

string. By using references, similar to pointers in C, data structures may be nested, creating arrays of hashes, for instance. The hash is a very powerful data structure. With it, we can easily store lists of information referenced by a string identifier. For instance, we can store a list of sectors in a hash, where each sector is accessed by its name. Each sector entry within this hash could point to another hash, which contains information about the sector: vertex lists, polygon lists, and portal lists. Scanning for overlapping portals could then be done by scanning through each portal of each other sector.

Furthermore, Perl supports object-oriented programming by providing for a class mechanism allowing for data abstraction, inheritance, and dynamic binding. This means we can write truly object-oriented programs with Perl.

Another feature that distinguishes Perl from similar languages is that it allows you to write scalable code. It doesn't do this for you automatically, but by sticking to standard object-oriented practices, you will find that any sort of problem which can be viewed as text processing of some sort is achieved well with Perl. This holds true for larger problems as well. For instance, one of the most stable VRML browsers for Linux, FreeWRL, is written in Perl; the Perl code interprets the textual VRML file, then interfaces with OpenGL to produce a real-time, dynamically alterable view of the virtual world. As you can imagine, this is no small undertaking. Because of its text processing slant, it may be tempting to compare Perl with other text processing languages, such as awk or a normal shell script written in the language of the shell interpreter (for instance, bash or csh). However, the data and control structures in awk or shell languages are severely lacking when it comes to larger programs. If you have ever attempted to write a large system, say more than 1,000 lines spread out over several files, in awk or a shell script, you will certainly understand what I mean: data structures have to be flattened out into strings, nesting data structures requires much trickery or maintenance of parallel data structures, temporary files need to be created to store some sorts of data, and so forth. With such languages, maintenance simply becomes a nightmare once the complexity of the system rises; the languages are not scalable. By the time you realize this, it's usually too late—you've already invested too much time in developing the system, and suddenly you have run up against the inherent limitations of the system. From personal experience, I have not yet encountered any limits or run up against any "walls" in Perl where the language itself has hindered further progress, whereas I frequently encountered such problems with other scripting languages.

Perl modules are another big plus. Very many Perl modules already exist which perform countless tasks. This allows you to build your Perl systems in a component-wise fashion. For instance, Perl scripts can easily interface with TCP/IP networks, making Perl ideal for WWW work, where most of the major protocols are in text (modules also exist for handling encrypted traffic over SSL). Also, as indicated above, there are interfaces from Perl to OpenGL, for doing graphics programming. The list of categories of available Perl modules is simply too long to include here. There's a well-organized network of Perl modules called CPAN (the comprehensive Perl archive network) which is mirrored at hundreds of sites on the Internet. These modules were written by programmers just like ourselves and submitted to CPAN for the benefit of the Perl community at large. If you want a parser for any text file format or protocol which is reasonably well

known, you can bet that there is probably already a Perl module on CPAN which parses it for you. CPAN is accessible under the URL `http://www.perl.com/CPAN`.

A final factor which makes Perl attractive is its excellent, comprehensive online documentation in the form of manual pages. Ranging from philosophy and basic concepts, to tutorials on data structures and object-oriented programming, to comprehensive reference information, it's all available via `man`, making incremental learning of Perl a realistic option.

With all of this praise, you might be wondering if there are any disadvantages to using Perl. Of course there are; no language is perfect. Some aspects which you might consider before using Perl are its inherent lack of type safety, its sometimes cryptic syntax, and its lack of ISO standardization. You should also check the availability of Perl tools and Perl libraries for the particular task you are trying to achieve before committing to Perl for a project, but the same holds true of any programming language. If another language has better tool and library support for your problem domain, then you might not want to use Perl, or only link in small Perl modules.

To summarize, here are the benefits of Perl:

- Excellent framework for dealing with text
- Support for object-orientation
- Scalability
- Good online documentation
- Active community support (CPAN) for a broad range of problem domains

Since our Videoscape files are text, and since our output world file is also text, the Videoscape-to-world-file conversion process, including the most significant phase, the identification of portal connectivity, can all be viewed as a text manipulation problem. We must parse the text Videoscape files, extract portal connectivity from the files, then write a world file as output.

With this understanding, let's take a look at the specific Perl modules comprising the system.

Architecture of the Perl Portalization System

The Perl portalization system consists of a number of scripts which can be divided into three categories: structural modules, parsing and generator modules, and controlling scripts. The next sections cover each of these categories individually.

We unfortunately don't have the space here to provide a tutorial on the Perl language. Check the manual page for Perl (type **man perl**) for extensive information on the various reference material and tutorials available online. We'll go over the major concepts of each Perl script at a high level so you can understand how they work, and also comment briefly on any particularly unusual Perl constructs used to achieve the goals.

The operation of the entire system can be understood as follows. Also, refer back to the data flow diagram of the system, Figure 6-10.

1. Parse all Videoscape files. Store all of the geometry in internal Perl data structures, organized by sector.
2. For each sector, scan the mesh for free edges. Link all free edges into continuous loops. Add each continuous loop as a portal polygon belonging to the sector.

3. For each sector and portal, scan all other sectors for an overlapping portal. If such an overlapping portal is found, link the original portal to the target sector containing the overlapping portal. If no such portal is found, the portal has no target sector, which is a modeling error; thus, delete the portal.
4. Print each sector's vertex, polygon, and portal definitions to the world file.

Keep the high-level operation of the system in mind as you read the following sections on the specific Perl modules used.

Structural Modules

The structural Perl modules in the portalization system store the geometrical data extracted from the Videoscape files in a structured manner, so that this information may be easily queried and changed. Each Perl module is essentially a class, in the object-oriented sense of the word. The organization of the structural classes is similar to that of the l3d classes. At the lowest level, we have 3D vertices (represented by module `Vertex.pm`). Based on lists of these vertices, we can build facets (polygons) and portals (modules `Facet.pm` and `Portal.pm`). A facet can contain a reference to a texture (`Texture.pm`). Facets and polygons are grouped into sectors (`Sector.pm`). A sector can also contain any number of actors (`Actor.pm`), which are plug-in objects. All sectors and actors belong to the world (`World.pm`).

A description of each of the structural modules follows.

Vertex.pm

Module `Vertex.pm` represents a vertex in 3D space.

Method `new` is the object constructor. It creates a new Perl hash object—which, remember, is just a list with members indexed by string—and inserts several items into the hash. These items are essentially the member variables of the class, and we will refer to them as such. In other words, we are using a Perl hash as an instance of a class, and are using elements of the hash object to represent member variables for that instance. This idea is identical for all of the Perl modules used in the portalization system.

Member variables `_x`, `_y`, and `_z` are the spatial coordinates of the vertex. External users of this class do not access the variables directly, but instead go through the member functions `x`, `y`, and `z`. Without an argument, these functions return the corresponding value; with an argument, they set the value to the given value. Another programming idiom, also used in the Perl portalization modules, is to have separate get and set routines, for instance `get_x` and `set_x`. All of the Perl portalization modules use this scheme of private hash members being accessed through member functions (either separate get/set methods or a unified get-set method), so we won't explicitly mention it again.

Method `dot` takes a second vertex as a parameter, and takes the dot product of itself and the given vertex. Both vertices are interpreted as the tips of vectors starting at the origin. This implies that the Perl `Vertex` class as defined here does not distinguish between vectors and points—a small sin, but if you've made it this far in the book, you hopefully know well the difference between vectors and points so as to understand which is meant within the context of a particular usage.

Method `dif` takes a second vertex as a parameter, and returns the vector difference between the location of the current vertex and the location of the second vertex—in other words, the vector from the second vertex to the current vertex.

Listing 6-1: `Vertex.pm`

```
package Vertex;
use strict;

############################
## the object constructor ##
############################
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{_x}  = undef;
    $self->{_y}= undef;
    $self->{_z} = undef;
    bless ($self, $class);
    return $self;
}

##########################################
## methods to access per-object data        ##
##                                        ##
## With args, they set the value.  Without  ##
## any, they only retrieve it/them.         ##
##########################################

sub x {
    my $self = shift;
    if (@_) {$self->{_x}= shift }

    return $self->{_x};
}

sub y {
    my $self = shift;
    if (@_) {$self->{_y}= shift }

    return $self->{_y};
}

sub z {
    my $self = shift;
    if (@_) {$self->{_z}= shift }

    return $self->{_z};
}

sub dot {
  my $self = shift;
  my $v2 = shift;

  return $self->x * $v2->x +
         $self->y * $v2->y +
         $self->z * $v2->z ;
}
```

```
sub dif {
  my $self = shift;
  my $v2 = shift;

  my $vnew = Vertex->new();
  $vnew->x ( $self->x - $v2->x );
  $vnew->y ( $self->y - $v2->y );
  $vnew->z ( $self->z - $v2->z );

  return $vnew;
}

1;  # so the require or use succeeds
```

Facet.pm

Module `Facet.pm` represents a polygonal facet. It stores the geometrical definition of the facet in terms of indices into a vertex list. It also provides methods which we need for the portalization process: accessing the facet's edge list, checking for polygon overlap, and using the facet as a texture space.

Method `new` is the object constructor. As a parameter, it expects a sector object, which is the parent of the facet object. In other words, facets cannot exist in isolation; they belong to a sector.

Member variable `_is_invisible` is an integer flag indicating if the facet is invisible (1) or visible (0). As we see later, we can use invisible facets to help define texture spaces to map textures to the walls of our sectors.

Member variable `_tex_id` is a string indicating which texture should be applied to this facet. Again, we cover texturing later in this chapter; also see module `Texture.pm`.

Member variable `_parent_sector` is a reference to the sector containing this facet, and is set in the constructor.

Member variable `_vertex_indices` is an array of integers which are indices into the parent sector's vertex list. These vertex indices define the polygon.

Method `get_edge_list` constructs a list of edges for the facet and returns it to the caller. An edge is simply a consecutive pair of vertex indices. The purpose of this method is to explicitly store all such pairs in a list, and to return this list. For instance, if a polygon is defined by the three vertex indices 0 7 2, then the edge list would be [[0,7] , [7,2], [2,0]]. Accessing the edges plays an important role in finding the free edges, which form the boundaries of portals.

Method `contains_edge` tells the caller whether or not the facet contains a particular edge, as specified by its two vertex indices. We use this method when scanning for free edges; if an edge is contained only by one facet, then it is a free edge; otherwise, it is not.

Method `coincident_with_facet` tells the caller whether or not the facet overlaps exactly with another facet, passed as a parameter. This is used in scanning for overlapping portals to link them together. It works by seeing if for each vertex in the original facet, a corresponding vertex at the same spatial location can be found in the other facet. If so, then the facets are coincident; otherwise, they are not. During comparison, we introduce an epsilon parameter to allow nearby points also to count as overlapping, to compensate for slight numerical or modeling errors.

Method `get_tex_ouv` interprets a triangular facet as a texture space specification and returns a hash with three elements representing the origin of the texture space, the tip of the *u* axis,

and the tip of the *v* axis. Methods `get_tex_origin`, `get_tex_u_vec`, and `get_tex_v_vec` call method `get_tex_ouv` and just return the appropriate single element requested (the origin, *u* tip, or *v* tip, respectively). The idea behind these methods is that a single triangular facet can be interpreted as a texture space; we call such a triangle a *texture definition triangle*. In order for this to work, the facet must be a right triangle—in other words, one of the angles must be exactly 90 degrees. The polygon vertex ordering determines the front/back orientation of the polygon; the left-handed convention we have been using throughout this book is that the front side is the side from which the polygon vertices appear clockwise. Then, we can define the triangle texture space as follows: when looking at the front side of the right triangle, the vertex on the 90 degree angle is the origin, the next vertex in the clockwise direction is the tip of the *u* axis, and the remaining vertex is the tip of the *v* axis. By appropriately scaling and positioning a triangle, we can then specify any texture space with a single polygon; we later see exactly how to do this in Blender, and how to flag a triangle so that it is recognized by the portalization system as being a texture definition triangle. Assuming a triangle has been found to have been marked as a texture definition triangle, the method `get_tex_ouv` then scans the triangle to find the vertex on the 90 degree angle, using the dot product between the two adjoining edges. This point is the origin of texture space; the next clockwise point is the *u* tip, and the other point is the *v* tip.

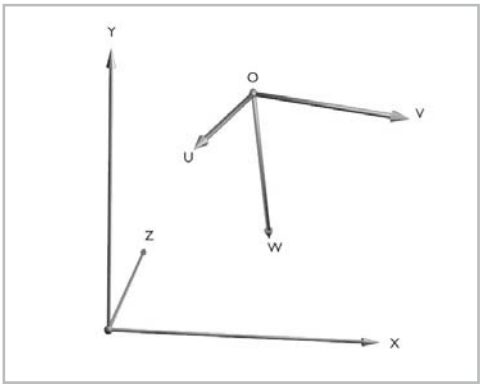


Figure 6-39: A texture space defined in world coordinates.

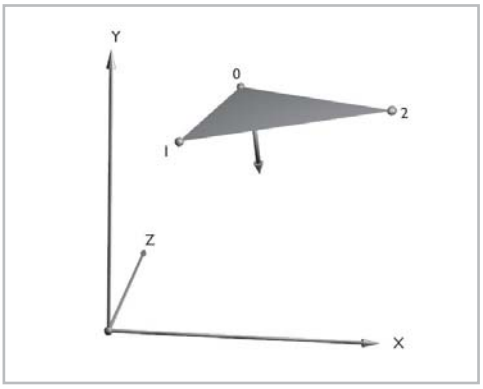


Figure 6-40: The texture space represented by a right triangle. Point 0 is on a 90 degree angle between two edges, and is thus the origin of the texture space. Point 1 is the next clockwise vertex (we are looking at the back side of the triangle), and is thus the tip of the u axis. Point 2 is the tip of the v axis. The w axis is not explicitly defined and is not needed for 2D textures.



NOTE Note that a texture space defined by means of a right triangle only specifies the location of the origin, the *u* axis tip, and the *v* axis tip. It does not allow for specification of a *w* axis. Specification of a *w* axis is only important if we use 3D volumetric textures (see Chapter 3), which we do not in this book.

Listing 6-2: Facet.pm

```
package Facet;
use strict;

my $debug = undef;

############################
## the object constructor ##
############################
sub new {
    my $proto = shift;

    my $parent = shift;

    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{_is_invisible}  = 0;
    $self->{_tex_id}  = "";
    $self->{_vertex_indices}  = [];
    $self->{_parent_sector}  = $parent;
    bless ($self, $class);
    return $self;
}

##########################################
## methods to access per-object data    ##
##                                      ##
## With args, they set the value.  Without  ##
## any, they only retrieve it/them.     ##
##########################################

sub get_is_invisible {
  my $self = shift;
  return $self->{_is_invisible};
}

sub set_is_invisible {
  my $self = shift;
  my $val = shift;
  $self->{_is_invisible}= $val;
}

sub get_tex_id {
  my $self = shift;
  return $self->{_tex_id};
}

sub set_tex_id {
  my $self = shift;
  my $id = shift;
  $self->{_tex_id}= $id;
}

sub get_parent_sector {
```

TEAMFLY

```
  my $self = shift;
  return $self->{_parent_sector};
}

sub set_parent_sector {
  my $self = shift;
  my $parent = shift;
  $self->{_parent_sector}= $parent;
}

sub get_vertex_index {
    my $self = shift;
    my $position = shift;

    return $self->{_vertex_indices}[$position];
}

sub get_vertex_index_count {
    my $self = shift;
    return $#{$self->{_vertex_indices}};
}

sub add_vertex_index {
    my $self = shift;
    my $index = shift;

    push @{$self->{_vertex_indices}},  $index;
}

#sub get_vertex_index_list {
#    my $self = shift;
#
#    return @{$self->{_vertex_indices}};
#}

sub get_edge_list {
  my $self = shift;

  my $i;
  my @edge_list;
  for($i=0; $i <= $self->get_vertex_index_count(); $i++) {
    my $j;
    $edge_list[$i] = [ $self->get_vertex_index($i) ];
    $j = $i + 1;
    if($j > $self->get_vertex_index_count()) {
      $j = 0;
    }
    push @{$edge_list[$i]}, $self->get_vertex_index($j);
  }
  return @edge_list;
}

sub contains_edge {
  my $self = shift;
  my $idx0 = shift;
  my $idx1 = shift;

  my $found = undef;

  my @edge_list = $self->get_edge_list();
```

```
  for( my $i=0; ( $i <= $#edge_list ) && ( !$found ); $i++ ) {
    if ( ( ( $edge_list[$i][0] == $idx0 ) && ( $edge_list[$i][1] == $idx1 ) )
       ||( ( $edge_list[$i][0] == $idx1 ) && ( $edge_list[$i][1] == $idx0 ) ) )
    {
      $found = 1;
    }
  }

  return $found;
}

sub coincident_with_facet {
  my $self = shift;
  my $epsilon = 1.001;

  my $other_facet = shift;
  my $result = 1;

  for ( my $i = 0; ($i <= $self->get_vertex_index_count()) && $result; $i++ ) {
    my $vertex_found_in_other = undef;
    for ( my $other_i = 0;
          ($other_i <= $other_facet->get_vertex_index_count())
          && !$vertex_found_in_other;
          $other_i++)
    {
      my $parent = $self->get_parent_sector();
      my $other_parent = $other_facet->get_parent_sector();
      if(
      (abs($parent->get_vertex($self->get_vertex_index($i)) -> x() -
           $other_parent->get_vertex
                          ($other_facet->get_vertex_index($other_i))->x())
          < $epsilon)
      &&(abs($parent->get_vertex($self->get_vertex_index($i)) -> y() -
          $other_parent->get_vertex
                           ($other_facet->get_vertex_index($other_i))->y())
          < $epsilon)
      &&(abs($parent->get_vertex($self->get_vertex_index($i)) -> z() -
          $other_parent->get_vertex
                           ($other_facet->get_vertex_index($other_i))->z())
          < $epsilon)
      )
      {
        $vertex_found_in_other = 1;
      }
    }
    if ( ! $vertex_found_in_other ) {
      $result = undef;
    }
  }

  return $result;
}

sub get_tex_ouv() {
  my $self = shift;

  my $v1, my $v2, my $v3;
  my $epsilon = 0.001;

  $v1 = Vertex->new();
```

```
  $v2 = Vertex->new();
  $v3 = Vertex->new();

  $v1->x ( $self->get_parent_sector()->get_vertex
             ($self->get_vertex_index(0)) -> x );
  $v1->y ( $self->get_parent_sector()->get_vertex
             ($self->get_vertex_index(0)) -> y() );
  $v1->z ( $self->get_parent_sector()->get_vertex
             ($self->get_vertex_index(0)) -> z() );
  $v2->x ( $self->get_parent_sector()->get_vertex
             ($self->get_vertex_index(1)) -> x() );
  $v2->y ( $self->get_parent_sector()->get_vertex
             ($self->get_vertex_index(1)) -> y() );
  $v2->z ( $self->get_parent_sector()->get_vertex
             ($self->get_vertex_index(1)) -> z() );
  $v3->x ( $self->get_parent_sector()->get_vertex
             ($self->get_vertex_index(2)) -> x() );
  $v3->y ( $self->get_parent_sector()->get_vertex
             ($self->get_vertex_index(2)) -> y() );
  $v3->z ( $self->get_parent_sector()->get_vertex
             ($self->get_vertex_index(2)) -> z() );

  # try origin at v1

  if ( $v2->dif($v1)->dot($v3->dif($v1)) < $epsilon) {
    if ($debug) {print "origin v1"; }
    return {o => $v1,
             v => $v3,
             u => $v2 };

  }

  # try origin at v2

  if ( $v1->dif($v2)->dot($v3->dif($v2)) < $epsilon) {
    if ( $debug )  {print "origin v2"; }
    return {o => $v2,
             v => $v1,
             u => $v3 };
  }

  # try origin at v3

  if ( $v1->dif($v3)->dot($v2->dif($v3)) < $epsilon) {
    if ( $debug ) {print "origin v3"; }
    return {o => $v3,
             v => $v2,
             u => $v1 };
  }

  else {
    if ( $debug ) { print "Hm... no origin found!" };
    print "WARNING: Invalid texture definition triangle found";
    print " (orientation could not be determined).";
  }

}

sub get_tex_origin() {
```

```
  my $self = shift;

  return $self->get_tex_ouv()->{'o'};

}

sub get_tex_u_vec() {
  my $self = shift;
  return $self->get_tex_ouv()->{'u'};
}

sub get_tex_v_vec() {
  my $self = shift;
  return $self->get_tex_ouv()->{'v'};
}

1;  # so the require or use succeeds
```

Texture.pm

Module `Texture.pm` represents a texture image and orientation. It stores a texture space, the name of a texture image file, and a unique string which identifies the texture object itself. Textures can be assigned to facets, as we mentioned earlier. Importantly, notice that assigning a `Texture` to a `Facet` not only assigns the image to the facet, but also the orientation defined in the `Texture`.

Member variable `_tex_id` is a string identifying this object. Its purpose is to allow us to define a single texture image and orientation, and assign a name to it, such as `stony_ceiling`. This way, we can set all polygons located in the ceiling of a room to use the texture `stony_ceiling`, which means that they all share the same texture image and the same texture space. This allows all polygons forming the ceiling, no matter how oddly shaped they are, to line up seamlessly on the edges, since they all share the same texture space.

Member variables `_u_vec`, `_v_vec`, and `_origin` define the texture space in terms of the tip of the *u* axis, the tip of the *v* axis, and the texture space origin.

Member variable `_image` is a string containing the filename of the texture image associated with this texture object.

Listing 6-3: `Texture.pm`

```
package Texture;
use strict;

use Portals::Vertex;

############################
## the object constructor ##
############################
sub new {
    my $proto = shift;

    my $parent = shift;

    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{_tex_id}  = "";
    $self->{_u_vec}  = Vertex->new();
```

```
    $self->{_v_vec}  = Vertex->new();
    $self->{_origin}  = Vertex->new();
    $self->{_image}  = "";
    bless ($self, $class);
    return $self;
}

##########################################
## methods to access per-object data    ##
##                                      ##
## With args, they set the value.  Without  ##
## any, they only retrieve it/them.     ##
##########################################

sub get_tex_id {
  my $self = shift;
  return $self->{_tex_id};
}

sub set_tex_id {
  my $self = shift;
  my $id = shift;
  $self->{_tex_id}= $id;
}

sub get_u_vec {
  my $self = shift;
  return $self->{_u_vec};
}

sub set_u_vec {
  my $self = shift;
  my $vec = shift;
  $self->{_u_vec}= $vec;
}

sub get_v_vec {
  my $self = shift;
  return $self->{_v_vec};
}

sub set_v_vec {
  my $self = shift;
  my $vec = shift;
  $self->{_v_vec}= $vec;
}

sub get_origin {
  my $self = shift;
  return $self->{_origin};
}

sub set_origin {
  my $self = shift;
  my $o = shift;
  $self->{_origin}= $o;
}

sub get_image {
  my $self = shift;
```

```
  return $self->{_image};
}

sub set_image {
  my $self = shift;
  my $i = shift;
  $self->{_image}= $i;
}

1;  # so the require or use succeeds
```

Portal.pm

Module `Portal.pm` represents a portal polygon. It inherits from the `Facet` class.

Member variable `_target_sector` (which, incidentally, is not explicitly created in the constructor, but is instead dynamically created upon assignment via method `set_target_sector`) is a reference to a sector object, to which the portal polygon leads.

Listing 6-4: `Portal.pm`

```
package Portal;

use Portals::Facet;
@ISA = qw(Facet);

sub get_target_sector {
  my $self = shift;
  return $self->{_target_sector};
}

sub set_target_sector {
  my $self = shift;
  my $target = shift;
  $self->{_target_sector}= $target;
}

1;
```

Sector.pm

Module `Sector.pm` represents a sector containing both geometric polygons and portal polygons leading to other sectors. It contains the logic for generating portals based on the holes in the sector geometry.

Member variable `_name` is a unique string identifier for this sector.

Member variable `_vertices` is a list of vertices of type `Vertex`. The vertices are accessible by an integer index indicating their position in the list.

Member variable `_facets` is a list of geometric polygons in the sector mesh. The polygons are of type `Facet`.

Member variable `_portals` is a list of portal polygons in the sector mesh and are of type `Portal`.

Member variable `_nonlooped_portals` is a list of incomplete portal polygons which could not be completely formed because the free edges did not form a closed loop, which is necessary for a valid portal polygon. This list can be printed for error checking purposes.

Member variable `_nonlinked_portals` is a list of invalid portal polygons for which no corresponding overlapping portal could be found, meaning that the portal could not be linked to any other sector.

Member variable `_free_edges` is a list of all free edges among all polygons in the sector mesh. It is filled through private method `_find_free_edges`.

Member variable `_actors` is a list of all actor objects (plug-in objects) which have been initially assigned to this sector. We see later how to use Blender to specify the location of actors.

Member variables `_node_min_{x,y,z}` and `_node_max_{x,y,z}` store an axially aligned bounding box for the sector, in terms of the minimum and maximum corner points.

Member variable `_has_geometry` indicates whether or not the sector contains any geometry. Normally it will return a true value; however, certain automatic sector generation schemes (discussed at the end of this chapter) can result in the creation of empty sectors containing no geometry, in which case this variable returns a false value.

Private method `_find_free_edges` is the first part of the portal generation process. It scans for free edges among all polygons within the sector. The logic is as follows. For each polygon, we loop through all of its edges. For each edge, we see if any other polygon edge in the sector contains the same edge. If so, the edge is used by more than one polygon, and cannot be a free edge. If not, the edge is only used by one polygon, and is thus a free edge. After execution of this method, member variable `_free_edges` contains the free edge list.

Private method `_merge_free_edges_into_portals` is the second part of the portal generation process. Given the list of free edges, we then proceed to find continuous loops of free edges. Each such loop is a portal. We proceed simply by starting with one free edge, which is a pair of vertices. Then, we try to daisy chain to the next free edge, by finding another free edge whose starting vertex is the same as the ending vertex of the current free edge. In this manner, we build up a loop, free edge by free edge. If we finally arrive at the starting free edge again, we have completed the loop, and can add it to the sector's list of portals. If any free edge loop can be started but not completed (i.e., if we do not finally arrive back at the starting free edge), then the input data was partially erroneous, and we have a non-looped portal.

Public method `make_portals_from_holes` simply calls `_find_free_edges` and `_merge_free_edges_into_portals` in sequence, and is the interface for external users to create portal polygons for this sector.

Listing 6-5: `Sector.pm`

```
package Sector;
use strict;
use Portals::Facet;
use Portals::Portal;
use Portals::Vertex;

############################
## the object constructor ##
############################
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{_name}  = undef;
```

```
    $self->{_vertices}= {};
    $self->{_facets} = {};
    $self->{_portals} = {};
    $self->{_actors}= {};
    $self->{_nonlooped_portals}= {};
    $self->{_nonlinked_portals}= {};
    $self->{_free_edges} = [];
    $self->{_has_geometry}= undef;
    $self->{_node_min_x}=999999;
    $self->{_node_min_y}=999999;
    $self->{_node_min_z}=999999;
    $self->{_node_max_x}=-999999;
    $self->{_node_max_y}=-999999;
    $self->{_node_max_z}=-999999;

    bless ($self, $class);
    return $self;
}

##############################################
## methods to access per-object data        ##
##                                          ##
## With args, they set the value.  Without  ##
## any, they only retrieve it/them.         ##
##############################################

sub node_min_x {
    my $self = shift;
    if (@_) {$self->{_node_min_x}= shift }

    return $self->{_node_min_x};
}

sub node_min_y {
    my $self = shift;
    if (@_) {$self->{_node_min_y}= shift }

    return $self->{_node_min_y};
}

sub node_min_z {
    my $self = shift;
    if (@_) {$self->{_node_min_z}= shift }

    return $self->{_node_min_z};
}

sub node_max_x {
    my $self = shift;
    if (@_) {$self->{_node_max_x}= shift }

    return $self->{_node_max_x};
}

sub node_max_y {
    my $self = shift;
    if (@_) {$self->{_node_max_y}= shift }

    return $self->{_node_max_y};
```

```
}

sub node_max_z {
    my $self = shift;
    if (@_) {$self->{_node_max_z}= shift }

    return $self->{_node_max_z};
}

sub has_geometry {
    my $self = shift;
    if (@_) {$self->{_has_geometry}= shift }

    return $self->{_has_geometry};
}

sub name {
    my $self = shift;
    if (@_) {$self->{_name}= shift }

    return $self->{_name};
}

sub get_all_vertices {
    my $self = shift;
    if (@_) {$self->{_vertices}= shift }

    return $self->{_vertices};
}

sub get_vertex {
    my $self = shift;
    my $vertex_index = shift;

    return $self->{_vertices}->{$vertex_index};
}

sub add_vertex {
    my $self = shift;
    my $vertex_index = shift;
    my $vertex = shift;
    $self->{_vertices}->{$vertex_index}= $vertex;
}

sub get_all_facets {
    my $self = shift;

    return $self->{_facets};

    #if (@_) {@{$self->{_facets}}= @_ }
    #return @{$self->{_facets}};
}

sub get_facet {
  my $self= shift;
  my $facet_index = shift;
  return $self->{_facets}->{$facet_index};
}

sub add_facet {
```

```
  my $self = shift;
  my $facet_index = shift;
  my $facet = shift;

  $self->{_facets}->{$facet_index}= $facet;
}

sub get_all_actors {
    my $self = shift;
    if (@_) {$self->{_actors}= shift }

    return $self->{_actors};
}

sub add_actor {
    my $self = shift;
    my $actor = shift;
    $self->{_actors}->{$actor->name()}= $actor;
}

sub get_actor {
    my $self = shift;
    my $key = shift;

    return $self->{_actors}->{$key};
}

sub get_all_portals {
    my $self = shift;
    if (@_) {$self->{_portals}= shift }

    return $self->{_portals};
}

sub get_portal {
  my $self= shift;
  my $portal_index = shift;
  return $self->{_portals}->{$portal_index};
}

sub add_portal {
  my $self = shift;
  my $portal_index = shift;
  my $portal = shift;

  $self->{_portals}->{$portal_index}= $portal;
}

sub delete_portal {
  my $self = shift;
  my $portal = shift;

  for my $a_index ( keys %{$self->get_all_portals()}) {
    if ($self->get_portal($a_index) == $portal) {
      print STDERR "Portal deleted.";
      delete $self->{_portals}->{$a_index};
    }
  }
}
```

TEAMFLY

```
sub get_free_edges {
    my $self = shift;

    return  $self->{_free_edges};
}

sub make_portals_from_holes {
  my $self = shift;

  printf(STDERR "Making portals from holes:");
  $self->_find_free_edges();
  $self->_merge_free_edges_into_portals();
}

sub add_nonlooped_portal {
    my $self = shift;
    my $portal_index = shift;
    my $portal = shift;

    $self->{_nonlooped_portals}{$portal_index}= $portal;
}

sub get_nonlooped_portal {
  my $self= shift;
  my $portal_index = shift;
  return $self->{_nonlooped_portals}->{$portal_index};
}

sub get_all_nonlooped_portals {
    my $self = shift;

    return $self->{_nonlooped_portals};
}

sub add_nonlinked_portal {
    my $self = shift;
    my $portal_index = shift;
    my $portal = shift;

    $self->{_nonlinked_portals}{$portal_index}= $portal;
}

sub get_all_nonlinked_portals {
    my $self = shift;

    return $self->{_nonlinked_portals};
}

sub get_nonlinked_portal {
  my $self= shift;
  my $portal_index = shift;
  return $self->{_nonlinked_portals}->{$portal_index};
}

##################### PRIVATE ######################

sub _find_free_edges {
  my $self = shift;

  my $aSector = $self;
```

```
  my $aFacet_key;
  my $aFacet;
  my $aOtherFacet_key;
  my $aOtherFacet;
  my @edge_list;

  print STDERR "Scanning for free edges, Sector ", $aSector->name(), "";
  for $aFacet_key ( keys % {$aSector->get_all_facets() }) {
    $aFacet = $aSector->get_facet($aFacet_key);
    print STDERR "  Facet ", $aFacet_key, "";
    @edge_list = $aFacet->get_edge_list();
    for ( my $i=0; $i<=$#edge_list; $i++) {
      my $edge_taken = undef;
      for $aOtherFacet_key ( keys % {$aSector->get_all_facets() }) {
        if($aOtherFacet_key != $aFacet_key) { # for all OTHER facets
          $aOtherFacet = $aSector->get_facet($aOtherFacet_key);
          if($aOtherFacet ->contains_edge($edge_list[$i][0],$edge_list[$i][1]))
          {
            $edge_taken = 1;
          }
        }
      }
      if ( !$edge_taken ) {
        print STDERR
              "    FREE EDGE ", $edge_list[$i][0]," ", $edge_list[$i][1], "";
        push @{$self->get_free_edges()},  $edge_list[$i] ;
      }
    }
  }
  print STDERR "  Summary of found free edges:";
  for ( my $i=0; $i<=$#{$self->get_free_edges()}; $i++) {
    print STDERR "  ";
    print STDERR ${@{$self->get_free_edges()}[$i]}[0]," ";
    print STDERR ${@{$self->get_free_edges()}[$i]}[1],"";
  }
}

sub _merge_free_edges_into_portals {
  my $self = shift;
  my $portal;

  # note that we must link the edges from pt1 to pt0 instead of pt0 to pt1
  # because we must reverse the orientation of the edges to make the portal
  # normal point the same direction as the neighboring facets' normals.

  my $portal_num = 0;
  my $unlooped_portal_num = 0;
  while ( ( $#{$self->get_free_edges()}>= 0 ) ) {

    my $error = undef;

    print STDERR "    Creating new portal.";
    $portal = Portal->new($self);
    $self->add_portal(sprintf("%d",$portal_num), $portal);

    my $e_idx = 0;
    my $idx1 = ${@{$self->get_free_edges()}[$e_idx]}[1];
    my $idx0 = ${@{$self->get_free_edges()}[$e_idx]}[0];

    # add current vertex to vertex list
```

```
$portal->add_vertex_index( $idx1 );
splice ( @{$self->get_free_edges()}, $e_idx, 1 ); # remove current edge
print STDERR "      Starting vertex $idx1.";

my $first_idx = $idx1;

while ( ( $idx0 != $first_idx ) && !$error ) {
  # keep going through vertex pairs until you hit the starting vertex again

  # find daisy-chained edge

  my $i;
  my $done = undef;

  for ( $i=0; ($i<=$#{$self->get_free_edges()}) && (!$done); $i++) {
    my $candidate_idx1 = ${@{$self->get_free_edges()}[$i]}[1];
    my $candidate_idx0 = ${@{$self->get_free_edges()}[$i]}[0];

    if ( $candidate_idx1 == $idx0 ) {
      $idx1 = $candidate_idx1;
      $idx0 = $candidate_idx0;

      # add current vertex to vertex list
      $portal->add_vertex_index( $idx1 );
      print STDERR "      Daisy-chaining vertex $idx1.";

      $e_idx = $i;
      # remove current edge
      splice ( @{$self->get_free_edges()}, $e_idx, 1 );
      $done = 1;
    }
    # the videoscape format appears to arbitrarily reverse
    # the order of vertices in two-vertex "faces" (probably since
    # no normal vector orientation can be computed based on two
    # vertices), so we have to also check for a matching edge in
    # the other direction, i.e. edge 0-1 and 1-3 daisy chains to 0-1-3,
    # but so does 0-1 and 3-1 (since 3-1 is the same as 1-3)
    elsif ( $candidate_idx0 == $idx0 ) {
      $idx1 = $candidate_idx0;
      $idx0 = $candidate_idx1;

      # add current vertex to vertex list
      $portal->add_vertex_index( $idx1 );
      print STDERR "      Daisy-chaining vertex $idx1.";

      $e_idx = $i;
      # remove current edge
      splice ( @{$self->get_free_edges()}, $e_idx, 1 );
      $done = 1;
    }
  }

  if ( !$done) {
    print STDERR
          "Uh oh, free edges couldn't be joined into a loop; skipping...";
    $self->delete_portal($portal);
    $self->add_nonlooped_portal(sprintf("%d",$unlooped_portal_num++),
                                $portal);
    $done = 1;
    $error = 1;
```

```
        }
      }
      if ( ! $error ) {
        print STDERR "    Portal complete.";
        $portal_num++;
      }
    }
  }

  1;  # so the require or use succeeds
```

Actor.pm

Module `Actor.pm` represents a plug-in object, or actor. Later in this chapter, we see how to specify actors in Blender.

Member variable `_name` is a string uniquely identifying the actor.

Member variable `_parent` is the sector object in which this actor initially resides; member variable `_parent_name` is the name of this sector object.

Member variable `_orientation` is a string containing the nine elements of the object's rotation matrix. A more flexible approach would store the rotation matrix as an array where each element is individually accessible, but in this case, we don't actually use the rotation matrix; we simply read it from the mesh, and rewrite it to the world file. This is why the orientation is stored as a simple string.

Member variable `_position` is a string containing the coordinates of the object's position. As with the orientation, the string storage format is used because we do not do any manipulation of the position.

Member variable `_is_camera` reflects whether or not the current plug-in object under consideration is the main camera in the scene. According to the world file format in Chapter 5, the camera is specified by a `CAMERA` line in the world file, whereas other plug-in objects are specified with an `ACTOR` line; thus, we need to differentiate between the two.

Member variable `_meshfile` is the name of the Videoscape file containing the geometry for this plug-in object. As we touched upon earlier and as we see later in more detail, we use simple cubes in Blender as placeholders for plug-in objects; the actual plug-in geometry is defined by this separate mesh file.

Member variable `_uvfile` is the name of the file containing the texture coordinates for the mesh.

Member variable `_texturefile` is the name of the file containing the texture image for the mesh.

Member variable `_pluginfile` is the name of the dynamic library file controlling the behavior of this plug-in object.

Listing 6-6: `Actor.pm`

```
package Actor;
use strict;
use Portals::Facet;
use Portals::Portal;
use Portals::Vertex;

############################
```

```
## the object constructor ##
############################
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{_name}  = undef;
    $self->{_parent}= undef;
    $self->{_parent_name}= undef;
    $self->{_orientation}= undef;
    $self->{_position} = undef;
    $self->{_is_camera} = undef;
    $self->{_meshfile} = undef;
    $self->{_uvfile}= undef;
    $self->{_texturefile}= undef;
    $self->{_pluginfile} = undef;

    bless ($self, $class);
    return $self;
}

##########################################
## methods to access per-object data    ##
##                                      ##
## With args, they set the value.  Without  ##
## any, they only retrieve it/them.     ##
##########################################

sub name {
    my $self = shift;
    if (@_) {$self->{_name}= shift }

    return $self->{_name};
}

sub parent_name {
    my $self = shift;
    if (@_) {$self->{_parent_name}= shift }

    return $self->{_parent_name};
}

sub parent {
    my $self = shift;
    if (@_) {$self->{_parent}= shift }

    return $self->{_parent};
}

sub is_camera {
    my $self = shift;
    if (@_) {$self->{_is_camera}= shift }

    return $self->{_is_camera};
}

sub meshfile {
    my $self = shift;
    if (@_) {$self->{_meshfile}= shift }
```

```
    return $self->{_meshfile};
}

sub uvfile {
    my $self = shift;
    if (@_) {$self->{_uvfile}= shift }

    return $self->{_uvfile};
}

sub texturefile {
    my $self = shift;
    if (@_) {$self->{_texturefile}= shift }

    return $self->{_texturefile};
}

sub pluginfile {
    my $self = shift;
    if (@_) {$self->{_pluginfile}= shift }

    return $self->{_pluginfile};
}

sub orientation {
    my $self = shift;
    if (@_) {$self->{_orientation}= shift }

    return $self->{_orientation};
}

sub position {
    my $self = shift;
    if (@_) {$self->{_position}= shift }

    return $self->{_position};
}

1;  # so the require or use succeeds
```

World.pm

Module `World.pm` is the highest-level structural module, which represents a portal world consisting of sectors and actors. Essentially, we parse all exported Videoscape files and store them into a `World` object, which we then traverse to print out the final portal world file.

Member variable `_sectors` is a list of sector objects, of type `Sector`, belonging to this world. For each Videoscape file, there is one sector object.

Member variable `_textures` is a list of all textures, of type `Texture`, available in the world. Remember that a `Texture` specifies both an image and an orientation.

Member variable `_texture_imagenums` is a unique, numbered list of all texture images referenced by all `Texture` objects in the `_textures` list. The idea is that if several `Texture` objects use the same texture image (but with different orientations), the texture image should not be loaded more than once. By collecting all texture images into a unique list, we prevent unnecessary duplication of texture images. We can reference each image in the unique texture image list

by a numerical identifier, which, not coincidentally, is also how the world file format refers to textures (by number).

Member variables `_min{x,y,z}` and `_max{x,y,z}` store the bounding box of all geometry in the world, in terms of minimum and maximum corner points. Member variables `_nodesize{x,y,z}` store predefined dimensions of a box. If we were using a regular spatial partitioning scheme (see the end of this chapter), we could divide the bounding box of all geometry into smaller boxes with dimensions specified by `_nodesize{x,y,z}`.

Method `link_portals` is the last part of automatic portal generation. Within the `Sector` class, we first found all free edges, then linked free edges into portals belonging to the sector. After doing this for all sectors, the last step is to find overlapping portals. For all sectors, for all portals, we must link each portal to the sector containing its overlapping counterpart. In other words, given a portal A in one sector, we scan for a portal B in another sector which exactly overlaps A. If we find such a portal B, then we link A to the sector containing B. If we find no such corresponding portal B, then this is an error, and A is a non-linked portal which should be deleted from its containing sector.

Listing 6-7: `World.pm`

```
package World;
use strict;
use Portals::Sector;
use Portals::Texture;

############################
## the object constructor ##
############################
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{_sectors}  = {};
    $self->{_textures}  = {};
    $self->{_texture_imagenums}  = {};
    $self->{_minx}  = 500000;
    $self->{_maxx}  = -500000;
    $self->{_miny}  = 500000;
    $self->{_maxy}  = -500000;
    $self->{_minz}  = 500000;
    $self->{_maxz}  = -500000;
    $self->{_nodesizex} = 200;
    $self->{_nodesizey} = 200;
    $self->{_nodesizez} = 200;

    bless ($self, $class);
    return $self;
}

##########################################
## methods to access per-object data    ##
##                                      ##
## With args, they set the value.  Without  ##
## any, they only retrieve it/them.     ##
##########################################

sub get_nodesizex {
```

```
    my $self = shift;
    return $self->{_nodesizex};
}

sub get_nodesizey {
    my $self = shift;
    return $self->{_nodesizey};
}

sub get_nodesizez {
    my $self = shift;
    return $self->{_nodesizez};
}

sub get_minx {
    my $self = shift;
    return $self->{_minx};
}

sub get_miny {
    my $self = shift;
    return $self->{_miny};
}

sub get_minz {
    my $self = shift;
    return $self->{_minz};
}

sub get_maxx {
    my $self = shift;
    return $self->{_maxx};
}

sub get_maxy {
    my $self = shift;
    return $self->{_maxy};
}

sub get_maxz {
    my $self = shift;
    return $self->{_maxz};
}

sub set_minx {
    my $self = shift;
    my $v = shift;
    $self->{_minx}= $v;
}

sub set_miny {
    my $self = shift;
    my $v = shift;
    $self->{_miny}= $v;
}

sub set_minz {
    my $self = shift;
    my $v = shift;
    $self->{_minz}= $v;
```

```
}

sub set_maxx {
    my $self = shift;
    my $v = shift;
    $self->{_maxx}= $v;
}

sub set_maxy {
    my $self = shift;
    my $v = shift;
    $self->{_maxy}= $v;
}

sub set_maxz {
    my $self = shift;
    my $v = shift;
    $self->{_maxz}= $v;
}

sub get_nodesizex {
    my $self = shift;
    return $self->{_nodesizex};
}

sub get_nodesizey {
    my $self = shift;
    return $self->{_nodesizey};
}

sub get_nodesizez {
    my $self = shift;
    return $self->{_nodesizez};
}

sub get_all_sectors {
    my $self = shift;

    return $self->{_sectors};
}

sub get_sector {
    my $self = shift;
    my $sector_name = shift;

    return $self->{_sectors}->{$sector_name};
}

sub add_sector {
    my $self = shift;
    my $sector_name = shift;
    my $sector = shift;

    $self->{_sectors}->{$sector_name}= $sector;
}

sub get_all_textures {
    my $self = shift;

    return $self->{_textures};
```

```
}

sub get_texture {
    my $self = shift;
    my $tex_id = shift;

    return $self->{_textures}->{$tex_id};
}

sub add_texture {
    my $self = shift;
    my $tex_id = shift;
    my $tex = shift;

    $self->{_textures}->{$tex_id}= $tex;
}

sub get_all_texture_imagenums {
    my $self = shift;

    return $self->{_texture_imagenums};
}

sub get_texture_imagenum {
    my $self = shift;
    my $texi_id = shift;

    return $self->{_texture_imagenums}->{$texi_id};
}

sub set_texture_imagenum {
    my $self = shift;
    my $texi_id = shift;
    my $texi = shift;

    $self->{_texture_imagenums}->{$texi_id}= $texi;
}

sub link_portals {
  my $self = shift;

  my $aSector_key;
  my $aSector;
  my $aPortal_key;
  my $aPortal;
  my $aOtherSector_key;
  my $aOtherSector;
  my $aOtherPortal_key;
  my $aOtherPortal;
  for $aSector_key ( keys % {$self->get_all_sectors() }) {
    my $unlinked_portal_num = 0;
    $aSector = $self->get_sector($aSector_key);

    print STDERR "Linking portals from Sector $aSector_key.";
    for $aPortal_key ( keys % {$aSector->get_all_portals() }) {
      $aPortal = $aSector->get_portal($aPortal_key);
      if ($aPortal->get_target_sector()) {
        print STDERR "  Already assigned other side of portal $aPortal_key";
        print STDERR "(Sector ", $aPortal->get_target_sector()->name(), ").";
      }else {
```

TEAMFLY

```
        print STDERR "  Searching for other side of portal $aPortal_key.";

        my $portal_linked = undef;
        for $aOtherSector_key ( keys % {$self->get_all_sectors() }) {
          if ($aOtherSector_key ne $aSector_key) {
            $aOtherSector = $self->get_sector($aOtherSector_key);
            for $aOtherPortal_key (keys %{$aOtherSector->get_all_portals()}) {
              $aOtherPortal = $aOtherSector->get_portal($aOtherPortal_key);
              if ( $aPortal->coincident_with_facet($aOtherPortal) ) {
                $portal_linked = 1;
                print STDERR "    Coincides with Sector $aOtherSector_key,";
                print STDERR " Portal $aOtherPortal_key";
                $aPortal->set_target_sector($aOtherSector);
                $aOtherPortal->set_target_sector($aSector); #reverse link
              }
            }
          }
        }
        if ( ! $portal_linked) {
          print STDERR "    No link found!";
          $aSector->delete_portal($aPortal);
          $aSector->add_nonlinked_portal(sprintf("%d",$unlinked_portal_num++),
                                         $aPortal);
        }
      }
    }
  }
}

1;  # so the require or use succeeds
```

Parsing and Generator Modules

We've now seen the Perl modules which store the structure of the entire portal world in memory. These structures form the middle stage of a pipeline. The input to the pipeline comes from a parser, which parses the Videoscape files and fills the world structures accordingly. On the output side, the world data is then fed through a generator, which traverses the world structure in memory and writes as output a valid portal world file to disk, conforming to the format specified in Chapter 5.

The next sections look at the input (parser) and output (generator) classes of the Perl portalization system.

VidscParser.pm

Module `VidscParser.pm` is a parser for the Videoscape file format. It only has one member variable, `_world`, which is the `World` object that will be filled by the parser.

Method `parse` reads all input files specified on the command line (which should be all Videoscape files forming the world), parses them to find the definitions of the geometric polygons, and stores these polygons in the world object. For now, you can ignore most of the `parse` routine, because most of it deals with the extraction of attribute information from the mesh, which we cover later in this chapter. The code which now interests us begins at the block marked "all untagged meshes are treated as sectors." Here, we create and fill a new sector object. We simply read in the facet definitions from the Videoscape file, change the coordinates from a right-handed to a left-handed system, and store these facets in the current sector object. We do this for all

Videoscape files, since one Videoscape file represents one sector. The next code block of interest then begins at the line marked "link portals together." At this point in the code, we have already read in and parsed all facet definitions for all sectors, and have stored these in memory. So, we call `make_portals_from_holes` for each sector, then call `link_portals` for the world to link all portals among all sectors. At this point, the sector objects in memory now contain both the geometry polygons and the portals, which are linked to their connected sectors. The data is ready to be written to disk by one of the generator classes, which we cover next.

Listing 6-8: `VidscParser.pm`

```
package VidscParser;

# defines variable $Bin, which is dir from which script was invoked
use FindBin;
use lib "$FindBin::Bin";

use strict;
use Portals::World;
use Portals::Actor;
use FileHandle;

my $debug = 0;
my $debug_str = "DEBUG----------";
my $portal_flag = hex '0x800000';
my $id_extraction_program = "$FindBin::Bin/../vidinfo/vidinfo";
my $attrib_filename_format_str = "attrib%d.dat";
my $attrib_dir = "";

############################
## the object constructor ##
############################
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{_world}  = World->new();
    bless ($self, $class);
    return $self;
}

##########################################
## methods to access per-object data    ##
##                                      ##
## With args, they set the value.  Without  ##
## any, they only retrieve it/them.     ##
##########################################

sub get_world {
    my $self = shift;

    return $self->{_world};
}

sub parse {

  my $self = shift;
```

```
my @files = @_;

my @actors;

for (my $file_idx=0; $file_idx <= $#files; $file_idx++) {
  my @mesh_properties;
  my $mesh_type;
  my $mesh_name;
  my $cmd = join(" ", $id_extraction_program, $files[$file_idx]);
  my @extracted_info = `$cmd`;
  my $extracted_id = $extracted_info[0];
  if($extracted_id) {
    my $attrib_file = sprintf($attrib_filename_format_str, $extracted_id);

    my @attrib_dir_tokens = split('/', $files[$file_idx]);
    pop @attrib_dir_tokens; # get path of current videoscape file
    $attrib_dir = join("/", @attrib_dir_tokens);
    if( length($attrib_dir) ) {
      $attrib_file = join("/",$attrib_dir, $attrib_file);
    }else {
      $attrib_file = $attrib_file;
    }
    my $fh = FileHandle->new();
    $fh->open($attrib_file);
    while(<$fh>) {
      push @mesh_properties, $_;
    }
    $fh->close();

    my @mesh_types = grep( /^TYPE /  , @mesh_properties);
    $mesh_type = $mesh_types[0];
    my @mesh_tokens = split(' ', $mesh_type);
    $mesh_type = $mesh_tokens[1];

    my @mesh_names = grep( /^NAME /  , @mesh_properties);
    $mesh_name = $mesh_names[0];
    my @meshname_tokens = split(' ', $mesh_name);
    $mesh_name= $meshname_tokens[1];
  }else {
    $mesh_type = undef;
    $mesh_name = undef;
  }

  if (lc($mesh_type) eq "actor") {
    my $actor = Actor->new();
    if(length($mesh_name)) {
      $actor->name($mesh_name);
    }else {
      $actor->name(sprintf("ACTOR%d",$file_idx));
    }

    my @mesh_matches;
    my $mesh_match;
    my @tokens;

    @mesh_matches = grep( /^PARENT /  , @mesh_properties);
    $mesh_match = $mesh_matches[0];
    @tokens = split(' ', $mesh_match);
    $mesh_match = $tokens[1];
    if($mesh_match) {
```

```
  $actor->parent_name($mesh_match);
}

@mesh_matches = grep( /^IS_CAMERA /  , @mesh_properties);
$mesh_match = $mesh_matches[0];
@tokens = split(' ', $mesh_match);
$mesh_match = $tokens[1];
if($mesh_match) {
  $actor->is_camera($mesh_match);
}

@mesh_matches = grep( /^MESH /  , @mesh_properties);
$mesh_match = $mesh_matches[0];
@tokens = split(' ', $mesh_match);
$mesh_match = $tokens[1];
if($mesh_match) {
  $actor->meshfile($mesh_match);
}

@mesh_matches = grep( /^TEXCOORDS /  , @mesh_properties);
$mesh_match = $mesh_matches[0];
@tokens = split(' ', $mesh_match);
$mesh_match = $tokens[1];
if($mesh_match) {
  $actor->uvfile($mesh_match);
}

@mesh_matches = grep( /^TEXTURE /  , @mesh_properties);
$mesh_match = $mesh_matches[0];
@tokens = split(' ', $mesh_match);
$mesh_match = $tokens[1];
if($mesh_match) {
  $actor->texturefile($mesh_match);
}

@mesh_matches = grep( /^PLUGIN /  , @mesh_properties);
$mesh_match = $mesh_matches[0];
@tokens = split(' ', $mesh_match);
$mesh_match = $tokens[1];
if($mesh_match) {
  $actor->pluginfile($mesh_match);
}

my $loc = $extracted_info[1];
my $or_row0 = $extracted_info[2];
my $or_row1 = $extracted_info[3];
my $or_row2 = $extracted_info[4];
$or_row0 =~ s/
$or_row1 =~ s/
$or_row2 =~ s/
$loc =~ s/

my @loc_parts = split(' ',$loc);
$loc = join(" ", $loc_parts[0], $loc_parts[2], $loc_parts[1]);

# to change orientation from rhs to lhs we must do two things:
# 1) swap the y and z columns in the matrix
# 2) swap the y and z rows in the matrix
my @row_parts = split(' ', $or_row0);
$or_row0 = join(" ", $row_parts[0], $row_parts[2], $row_parts[1]);
```

```
  @row_parts = split(' ', $or_row1);
  $or_row1 = join(" ", $row_parts[0], $row_parts[2], $row_parts[1]);
  @row_parts = split(' ', $or_row2);
  $or_row2 = join(" ", $row_parts[0], $row_parts[2], $row_parts[1]);

  $actor->position($loc);
  $actor->orientation(join(" ", $or_row0, $or_row2, $or_row1));
  push @actors, $actor;
}
else # elsif(lc($mesh_type) eq "sector")
{

  ######################################################################
  # all untagged meshes are treated as sectors
  ######################################################################

  my $sector = Sector->new();

  if(length($mesh_name)) {
    $sector->name($mesh_name);
  }else {
    $sector->name(sprintf("S%d",$file_idx));
  }
  $self->get_world()->add_sector($sector->name(), $sector);

  my %sectors;
  undef %sectors;

  open (FILE,$files[$file_idx]);
  if ($debug) {print "$debug_str $files[$file_idx] opened"; }

  my $id = <FILE>;
  my $vertex_count = <FILE>; # no. of vertices

  # read in vertex list from file

  for (my $i=0; $i<$vertex_count; $i++) {
    my $vertex = Vertex->new();
    $sector->add_vertex( sprintf("%d",$i), $vertex );
    my $line = <FILE>;

    my @list = split(' ',$line);

    # blender/videoscape uses a +z-up RHS system; we use a +y-up LHS system.
    $vertex->x($list[0]);
    $vertex->z($list[1]);
    $vertex->y($list[2]);
  }

  # read in facet definitions from file

  my @face_properties = grep( /^FACE /  , @mesh_properties);
  print STDERR "Mesh properties: -------";
  print STDERR @mesh_properties;
  print STDERR "Face properties: -------";
  print STDERR @face_properties;

  my $facet_number = 0;
  while ( <FILE> ) {
```

```
my @list = split(' ');

if( $list[0] < 3 ) {
  next; # this "polygon" has less than three edges: either it is
        # a special ID flag, or it is a modeling error, but it
        # certainly is not geometry we need to write to the final
        # world file
}

pop @list; # pop color off list
shift @list; # don't care about vertex index count

my $facet = Facet->new($sector);
$facet->set_is_invisible(0); # may be changed below
# we add the facet to the sector object LATER, after it is determined
# if it is visible or not (if not it is just there to define a
# texture orientation)

my @this_face_properties =
  grep( /^FACE *$facet_number /, @face_properties);

print STDERR @this_face_properties;

# we count the vertex indices in reverse because blender/videoscape
# normally store the vertices counterclockwise(rhs) whereas we use
# clockwise (rhs)

for(my $i=$#list; $i>=0; $i--) {
  $facet->add_vertex_index( $list[$i] );
}

my $tex_id;
my $tex_image_name;
my @matching_texspace_lines =
  grep( /_TEX_SPACE/ , @this_face_properties);
if($#matching_texspace_lines > -1) {
  # a polygon should EITHER use OR define a texture space, not both.
  # also, a polygon can't use or define multiple textures. so, we
  # just take the first matching line with _TEX_SPACE and extract the
  # texture id from it.
  my @matching_texspace_tokens =
    split(' ', $matching_texspace_lines[0]);

  # joining sector name with tex id name ensures uniqueness among
  # reused texture space names among sectors. not joining it with sector
  # name allows texture spaces to be reused across sectors, since
  # the texture space list is global in the world, not local in the sector.
  $tex_id = join(".", $sector->name(), $matching_texspace_tokens[3]);
  $tex_image_name = $matching_texspace_tokens[4];
}

if ( grep( /IS_TEX_SPACE/ , @this_face_properties )
&&   !grep( /IS_INVISIBLE/, @this_face_properties ) )
{
  if ($facet->get_vertex_index_count() != 2) {
    printf(STDERR
           "Non-triangle marked as geometry and tex-def-tri. ID: %s.",
           $tex_id);
  }else {
    printf(STDERR "Geometry and tex-def-tri found. ID: %s", $tex_id);
```

```
            $facet->set_tex_id($tex_id);

            $self->get_world()->set_texture_imagenum($tex_image_name, "dummy");
            # we just insert the image name as the hash KEY into the hash;
            # the value, which is the numerical index, is assigned later
            # in l3dGen after all tex images have been defined.
            # using a hash ensures uniqueness among the keys.

            my $tex = Texture->new();
            $tex->set_tex_id($tex_id);
            $tex->set_image($tex_image_name);
            $tex->set_origin($facet->get_tex_origin());
            $tex->set_u_vec($facet->get_tex_u_vec());
            $tex->set_v_vec($facet->get_tex_v_vec());
            $self->get_world()->add_texture(sprintf("%s",$tex_id), $tex);
          }
        }elsif ( grep( /IS_TEX_SPACE/ , @this_face_properties )
             &&   grep( /IS_INVISIBLE/, @this_face_properties ) )
        {
          if ($facet->get_vertex_index_count() != 2) {
            printf(STDERR
                   "Non-triangle marked as pure tex-def-tri. ID: %s.",
                   $tex_id);
          }else {
            printf(STDERR
                   "Pure tex-def-tri found. ID: %s", $tex_id);
            $facet->set_is_invisible(1); # face not to be written to world file
            $facet->set_tex_id($tex_id);

            $self->get_world()->set_texture_imagenum($tex_image_name, "dummy");
            # we just insert the image name as the hash KEY into the hash;
            # the value, which is the numerical index, is assigned later
            # in l3dGen after all tex images have been defined.
            # using a hash ensures uniqueness among the keys.

            my $tex = Texture->new();
            $tex->set_tex_id($tex_id);
            $tex->set_image($tex_image_name);
            $tex->set_origin($facet->get_tex_origin());
            $tex->set_u_vec($facet->get_tex_u_vec());
            $tex->set_v_vec($facet->get_tex_v_vec());
            $self->get_world()->add_texture(sprintf("%s",$tex_id), $tex);
          }
        }elsif ( grep( /USES_TEX_SPACE/ , @this_face_properties ) ) {
          printf(STDERR "Geometry poly num %d linked to tex ID: %s",
                 $facet_number, $tex_id);
          $facet->set_tex_id($tex_id);
        }else {
          printf(STDERR "Normal Geometry poly num %d", $facet_number);
        }

        # if facet was not invisible, add it to the world file
        if($facet->get_is_invisible() == 0) {
          $sector->add_facet( sprintf("%d",$facet_number), $facet );
        }
        $facet_number++;
      }
    }
  }
```

```
    # link portals together

    my $aSector_key;
    my $aSector;
    for $aSector_key ( keys % {$self->get_world()->get_all_sectors() }) {
      $aSector = $self->get_world()->get_sector($aSector_key);
      $aSector->make_portals_from_holes();
    }
    $self->get_world()->link_portals();

    # assign actors to sectors

    print STDERR "Now assigning ", $#actors + 1, " actors to their sectors.";
    my $i;
    for ( $i = 0; $i<=$#actors; $i++) {
      print STDERR "  Assigning actor ", $actors[$i]->name(), " to sector ";
      print STDERR $actors[$i]->parent_name(), "...";
      $aSector = $self->get_world()->get_sector
        ($actors[$i]->parent_name());
      if($aSector) {
        $aSector->add_actor($actors[$i]);
        printf(STDERR "done");
      }else {
        printf(STDERR
               " uh oh, couldnt't find parent sector... object discarded.");
      }
    }
  }

  1; # so the require or use succeeds
```

l3dGen.pm

Module `l3dGen.pm` is a generator class, which generates a portal world file based on the information stored in a `World` object.

Member variable `_world` stores the world object, which must have already been filled by a parser object.

Method `_generate` is a method which simply traverses the world structure in memory and prints the appropriate lines, conforming to the world file format of Chapter 5, to the standard output stream. The data in memory is already organized similarly to the format needed in the world file. The first thing we do is print a unique list of all texture images that are used by all `Texture` objects used by all `Facet` objects. Then we print a count of total sectors and actors, followed by sector and actor blocks. For each sector, we print all vertices and polygons. Non-textured geometry polygons (which are the only kind we have created thus far) cause generation of a `GEOMPOLY_TEX` line, which is a textured polygon, but the texture defaults to be the first texture in the world file, and the orientation is automatically computed based on the edges of the polygon. Explicitly textured geometry polygons generate a `GEOMPOLY_TEX` line with the specific image and orientation specified by the `Texture` object attached to the polygon. Portal polygons generate a `PORTALPOLY` line. After printing the polygons for the sector, we then print a list of actors assigned to the sector, in the form of `ACTOR` lines for normal plug-in objects or a `CAMERA` line for the camera object.

Listing 6-9: l3dGen.pm

```
package l3dGen;
use strict;
use Portals::World;

my $debug = 0;
my $debug_str = "DEBUG————————-";
my $portal_flag = hex '0x800000';

############################
## the object constructor ##
############################
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{_world}  = undef;
    bless ($self, $class);
    return $self;
}

##########################################
## methods to access per-object data     ##
##                                       ##
## With args, they set the value.  Without  ##
## any, they only retrieve it/them.      ##
##########################################

sub get_world {
    my $self = shift;

    return $self->{_world};
}

sub set_world {
    my $self = shift;
    my $w = shift;

    $self->{_world}= $w;
}

sub generate {
  my $self = shift;

  my $aSector_key;
  my $aSector;

  ###########################################################################
  # create the world file

  # texture images

  # assign a numerical id to each texture image, and print them in this
  # order; they will then be read back in in this order at runtime
  my $ikey;
  my $tex_image_num;
  $tex_image_num = 0;
  for $ikey ( sort keys %{$self->get_world()->get_all_texture_imagenums() }) {
    $self->get_world()->set_texture_imagenum($ikey, $tex_image_num);
```

```
  $tex_image_num++;
}
print $tex_image_num, " TEXTURES";
for $ikey ( sort keys %{$self->get_world()->get_all_texture_imagenums() }) {
  print $ikey, "";
}

# textures (orientations)

my %tex_name_to_num = {};

my $total_actors_and_sectors=0;
for my $aSector_key( keys % {$self->get_world()->get_all_sectors() }) {
  # +1 for this sector
  $total_actors_and_sectors++;

  # add in this sectors objects too
  $aSector = $self->get_world()->get_sector($aSector_key);
  for my $dummy2 ( keys % {$aSector->get_all_actors() }) {
    $total_actors_and_sectors++;
  }
}
print "$total_actors_and_sectors SECTORS_AND_ACTORS";

# sector info

for $aSector_key ( sort keys % {$self->get_world()->get_all_sectors() }) {
  $aSector = $self->get_world()->get_sector($aSector_key);

  print "SECTOR ", $aSector->name();
  if($aSector->has_geometry()) {
    printf(" %d %d %d %d %d %d ",
      $aSector->node_min_x() - $self->get_world()->get_minx(),
      $aSector->node_min_y() - $self->get_world()->get_miny(),
      $aSector->node_min_z() - $self->get_world()->get_minz(),
      $aSector->node_max_x() - $self->get_world()->get_minx(),
      $aSector->node_max_y() - $self->get_world()->get_miny(),
      $aSector->node_max_z() - $self->get_world()->get_minz());
  }
  print "";

  # vertices, facets
  my $num_vertices = 0;
  for my $v ( keys % {$aSector->get_all_vertices() }) {$num_vertices++; }
  print $num_vertices, " ";
  my $num_facets = 0;
  for my $f ( keys %{$aSector->get_all_facets()}) {$num_facets++; }
  my $num_portals = 0;
  for my $p ( keys %{$aSector->get_all_portals()}) {$num_portals++; }
  print $num_facets + $num_portals;
  print "";

  # vertices

  my $vertex_key;
  my $aVertex;
  for $vertex_key ( sort {$a<=>$b}(keys %{$aSector->get_all_vertices()}) ) {
    $aVertex = $aSector->get_vertex($vertex_key);
    print $vertex_key , " ";
    print $aVertex->x() , " ";
```

TEAM-FLY

```
  print $aVertex->y() , " ";
  print $aVertex->z() , "";
}

# facets

my $facet_key;
my $aFacet;

for $facet_key ( keys %{$aSector->get_all_facets()}) {
  $aFacet = $aSector->get_facet($facet_key);

  if($aFacet->get_is_invisible()) {next; }

  if ((length($aFacet->get_tex_id) == 0)
  || ( ! exists $self->get_world()->get_all_textures->
            {$aFacet->get_tex_id()}) )
  # could be falsely linked to a nonexistent texture id
  {
    print "GEOMPOLY ";
    print $aFacet->get_vertex_index_count()+1;
    for(my $i=0; $i<=$aFacet->get_vertex_index_count(); $i++) {
      print " " , $aFacet->get_vertex_index($i);
    }
    print "";
  }

  else {

   my $aTex = $self->get_world()->get_texture
     (sprintf("%s",$aFacet->get_tex_id()));

    print "GEOMPOLY_TEX "; # ,$aTex->get_tex_id()," ";
    print $aFacet->get_vertex_index_count()+1;
    for(my $i=0; $i<=$aFacet->get_vertex_index_count(); $i++) {
      print " " , $aFacet->get_vertex_index($i);
    }
    # texture image number
    # print " ", $tex_name_to_num{$aTex->get_tex_id()};
    print " ", $self->get_world()->get_texture_imagenum
                ($aTex->get_image());
    # texture orientation
    printf(" %f %f %f ",
      $aTex->get_origin()->x(),
      $aTex->get_origin()->y(),
      $aTex->get_origin()->z());
    printf(" %f %f %f ",
      $aTex->get_u_vec()->x(),
      $aTex->get_u_vec()->y(),
      $aTex->get_u_vec()->z());
    printf(" %f %f %f ",
      $aTex->get_v_vec()->x(),
      $aTex->get_v_vec()->y(),
      $aTex->get_v_vec()->z());
    print "";
  }

}
```

```
    # portals

    my $portal_key;
    my $aPortal;

    for $portal_key ( keys %{$aSector->get_all_portals()}) {
      $aPortal = $aSector->get_portal($portal_key);
      print "PORTALPOLY ";
      print $aPortal->get_vertex_index_count()+1;
      for(my $i=0; $i<=$aPortal->get_vertex_index_count(); $i++) {
        print " " , $aPortal->get_vertex_index($i);
      }
      print " ",$aPortal->get_target_sector()->name();
      print "";
    }

    # actors within this sector

    my $actor_key;
    for $actor_key ( keys %{$aSector->get_all_actors()}) {
      my $anActor;
      $anActor = $aSector->get_actor($actor_key);
      if ($anActor->is_camera()) {
        print "CAMERA ", $anActor->name(), " ";
        print "no_plugin ";
        print $anActor->position(), " ";
        print $anActor->orientation(), " ";
      }else {
        print "ACTOR ", $anActor->name(), " ";
        print $anActor->pluginfile(), " ";
        print $anActor->position(), " ";
        print $anActor->orientation(), " ";
        print $anActor->meshfile(), " ";
        print $anActor->texturefile(), " ";
        print $anActor->uvfile(), " ";
      }
      print "";
    }
  }
}
```

errGen.pm

Module `errGen.pm` is a generator class that attempts to identify errors in a `World` object and prints diagnostics as output.

Member variable `_world` stores the world object, which must have already been filled by a parser object.

Method `_generate` searches for errors in the world structure in memory, prints diagnostics as output, and creates a series of Videoscape files which visually display the location of the errors. The error checking looks at the non-looped and non-linked portal lists of each sector, prints a text message indicating the existence of such portals for the sector, and creates a Videoscape file containing edges illustrating the boundaries of such invalid portals. The output files are in `nonlinked.obj*` and `nonlooped.obj*`; the asterisk indicates that several sequentially numbered Videoscape files may be created, one for each sector with invalid portals. By loading these Videoscape files into Blender, you can see the spatial location of any invalid portals. By

storing the level meshes in one layer or scene and the invalid portals in another layer or scene, you can easily switch back and forth between the two to see exactly which sector meshes cause the invalid portals.

Listing 6-10: errGen.pm

```
package errGen;
use strict;
use Portals::World;
use FileHandle;

my $debug = 0;
my $debug_str = "DEBUG--------";
my $portal_flag = hex '0x800000';

############################
## the object constructor ##
############################
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{_world}  = undef;
    bless ($self, $class);
    return $self;
}

##########################################
## methods to access per-object data     ##
##                                       ##
## With args, they set the value.  Without  ##
## any, they only retrieve it/them.      ##
##########################################

sub get_world {
    my $self = shift;

    return $self->{_world};
}

sub set_world {
    my $self = shift;
    my $w = shift;

    $self->{_world}= $w;
}

sub generate {
  my $self = shift;

  my $aSector_key;
  my $aSector;

  my $unlinked_num;
  my $unlooped_num;

  # error report

  unlink <nonlinked.obj*>;
  unlink <nonlooped.obj*>;
```

```
for $aSector_key ( keys % {$self->get_world()->get_all_sectors() }) {
  my $portal_key;
  my $aPortal;

  $aSector = $self->get_world()->get_sector($aSector_key);

  if ( keys %{$aSector->get_all_nonlooped_portals}) {
    print STDERR "Sector ", $aSector_key, " has nonlooped portals.";

    # vertices

    my $fname;

    if ($unlooped_num) {
      $fname = sprintf("> nonlooped.obj%d", $unlooped_num++);
    }else {
      $fname = sprintf("> nonlooped.obj");
      $unlooped_num = 1;
    }

    my $fh = FileHandle->new();
    $fh->open($fname);
    print $fh "3DG1";

    my $num_vertices = 0;
    for my $v ( keys % {$aSector->get_all_vertices() }) {$num_vertices++; }
    print $fh $num_vertices,"";
    my $vertex_key;
    my $aVertex;
    for $vertex_key ( sort{$a<=>$b}keys %{$aSector->get_all_vertices()}) {
      $aVertex = $aSector->get_vertex($vertex_key);
      # print "(" , $vertex_key , ")";
      print $fh " " , $aVertex->x();
      print $fh " " , $aVertex->z(); # reverse y and z to conv. lhs<->rhs
      print $fh " " , $aVertex->y();
      print $fh "";
    }

    for $portal_key ( keys %{$aSector->get_all_nonlooped_portals()}) {
      $aPortal = $aSector->get_nonlooped_portal($portal_key);

      # since the portal may have more than 4 sides, and since Blender
      # only handles polygons with max 4 sides, we can't write out the
      # portal as a single polygon; instead we write it out edge by edge
      my $edge;
      for $edge ( $aPortal->get_edge_list() ) {
        print $fh "2 ",$edge->[0], " ", $edge->[1];
        print $fh " 0xFFFFFF";
      }
    }

    $fh->close();
  }
  if ( keys %{$aSector->get_all_nonlinked_portals}) {
    print STDERR "Sector ", $aSector_key, " has nonlinked portals.";

    # vertices

    my $fname;
```

```
    if ($unlinked_num) {
      $fname = sprintf("> nonlinked.obj%d", $unlinked_num++);
    }else {
      $fname = sprintf("> nonlinked.obj");
      $unlinked_num = 1;
    }

    my $fh = FileHandle->new();
    $fh->open($fname);
    print $fh "3DG1";

    my $num_vertices = 0;
    for my $v ( keys % {$aSector->get_all_vertices() }) {$num_vertices++; }
    print $fh $num_vertices,"";
    my $vertex_key;
    my $aVertex;
    for $vertex_key ( sort{$a<=>$b}keys %{$aSector->get_all_vertices()}) {
      $aVertex = $aSector->get_vertex($vertex_key);
      # print "(" , $vertex_key , ")";
      print $fh " " , $aVertex->x();
      print $fh " " , $aVertex->z(); # reverse y and z to conv. lhs<->rhs
      print $fh " " , $aVertex->y();
      print $fh "";
    }

    for $portal_key ( keys %{$aSector->get_all_nonlinked_portals()}) {
      $aPortal = $aSector->get_nonlinked_portal($portal_key);

      # since the portal may have more than 4 sides, and since Blender
      # only handles polygons with max 4 sides, we can't write out the
      # portal as a single polygon; instead we write it out edge by edge
      my $edge;
      for $edge ( $aPortal->get_edge_list() ) {
        print $fh "2 ",$edge->[0], " ", $edge->[1];
        print $fh " 0xFFFFFF";
      }
    }

    $fh->close();

  }
 }
}
```

Controlling Scripts

As is often the case with object-oriented systems, we have a number of components (classes), each of which represents a useful, coherent data abstraction in the problem domain. (Don't fall into the trap of thinking that a class does one thing well, because a class is a data abstraction, not a functional unit.) To bring the entire system together, we have a controlling script which creates the proper objects and sets them in motion. For our portalization system, this script is `vid2port.pl`.

vid2port.pl

Controlling script `vid2port.pl` executes the portalization system by creating and invoking methods on the appropriate structural objects.

Invoke `vid2port.pl` with a command of the form

```
perl $L3D/source/utils/portalize/vid2port.pl name.obj* >
world.dat
```

Replace `name.obj` with the name prefix of the Videoscape file to which you exported your meshes. Include the asterisk on the command line; this causes the shell to pass all Videoscape files with matching names as parameters to the Perl script. As output, this creates file `world.dat`; diagnostic messages are also printed to the standard error stream during the portalization process.

The script begins by creating an instance of a Videoscape parser object. It then calls `parse` on the parser object, passing a list of all the Videoscape filenames as a parameter. The parser object creates in memory a world object corresponding to the Videoscape files. Next, `vid2port.pl` creates an error generator object, assigns the world object to the generator, and calls `generate`. This creates the `nonlooped.obj*` and `nonlinked.obj*` Videoscape files to help find invalid portals. Finally, `vid2port.pl` creates an l3d generator object, assigns the world object to this generator, and calls `generate`. This prints the final portal world file to the standard output stream, which we redirected on the command line into file `world.dat`.

Listing 6-11: `vid2port.pl`

```
#!/usr/bin/perl

use FindBin;
use lib "$FindBin::Bin";

use Portals::VidscParser;
use Portals::l3dGen;
use Portals::errGen;
use strict;

my $parser = VidscParser->new();
$parser->parse(<@ARGV>);

my $g = errGen->new();
$g->set_world($parser->get_world());
$g->generate();

my $g = l3dGen->new();
$g->set_world($parser->get_world());
$g->generate();
```

Embedding Location, Orientation, Texture, Actor, and Other Information into Meshes

We now understand the basics of the portalization system: parsing Videoscape, scanning for holes, generating portals, linking portals, and creating the final world file as output. Now it's time to see how we can use the system to specify information on plug-in objects and textures. Doing so requires some fundamental enhancements to the system.

Notice that the entire portalization system so far has functioned without knowing which exported Videoscape file corresponded to which mesh in Blender. Up until now, this was

irrelevant. Portal and sector connectivity was determined in an object-independent manner by scanning for spatial overlap of portal polygons. If we found two overlapping portals, then we knew that the sector on one side—regardless of which one it actually was—was connected via a portal to the sector on the other side.

This situation changes as soon as we want to associate any specific information with a specific mesh. In particular, texture mapping and assignment of actors to sectors require us to uniquely identify meshes. For instance, in a sector representing a bathroom we might want to apply a tile texture to the walls, while in a living room sector we might use a wood texture. If we do not know which Videoscape file represents the bathroom and which represents the kitchen, we have no way of knowing which texture should be assigned to which mesh. We need a way of uniquely identifying each mesh before and after Videoscape export and a way of storing additional information, such as texture images and orientations, with each mesh.

The solution chosen here is to use overlapping edges in the geometry to encode a numerical identifier and an orientation within the mesh itself, and to use an external database to store additional information corresponding to each numerical ID. The next sections cover the system in detail.

Basic Ideas of Associating Attributes with Objects

The main idea we need to be aware of is the association of arbitrary attributes with a particular mesh. We call this system the attribute system. First, let's look at the basic ideas of the attribute system and the tools that implement these ideas. Then, we proceed on to an example of actually using the attribute system to specify texture and actor information for sectors.

Store an ID, Location, and Orientation in Overlapping Edges

The first goal is to be able to uniquely identify each mesh before and after export. The design of this system was dictated by the limitations of the Videoscape export system supported by Blender. As mentioned earlier, two major properties of this system make a solution difficult: only mesh geometry is exported, and we lose name information when exporting multiple meshes simultaneously.

This means that if we can somehow encode the mesh's identity within the geometry itself, then this information would indeed be preserved after Videoscape export. The minimum information needed to uniquely identify a mesh is a single integer ID number. Therefore, we need to find a way to encode an integer number into the mesh geometry, in such a way that it can be automatically extracted later for identification.

To accomplish this, we insert extra redundant polygons into the mesh geometry, where each polygon has just one edge (i.e., two vertices). Ordinarily, a one-edge polygon is a geometric impossibility, but we are using such polygons and edges to encode information, not represent geometry. These edges, and their vertices, encode an integer number. We call such edges *magic edges*, and their vertices *magic vertices*. Fortunately, such non-geometric edges are indeed exported by Blender, in their original order, into the Videoscape file, and remain preserved upon import.

It turns out that another important requirement, that of storing object location and orientation, can also be fulfilled by using such magic edges to specify a coordinate system. We thus solve both problems at once, as we see shortly. Storing object location and orientation is important for plug-in objects (less so for sectors), because each plug-in object should have a local object coordinate system which allows us to position and orient the object as a whole in world coordinates.

The system relies on two different positions of edges. One edge position represents a bit with the value zero; another edge position represents a bit with the value one. Given these two edge positions, we can then insert an arbitrary number of extra magic edges into the mesh, each edge having either the one position or the zero position. In this way, we can encode a binary number into the mesh geometry.The whole system works as follows.

1. Find the geometric center of the object by averaging all x, y, and z values. Call this center point (i,j,k).
2. Insert four exactly overlapping one-edge polygons into the geometry. Each of these edges goes from the center point (i,j,k) to point $(i+1,j,k)$. The presence of these four overlapping edges marks the beginning of a sequence of magic edges; it is sort of a start of transmission flag.
3. Insert a one-edge polygon into the geometry going from (i,j,k) to $(i+1,j,k)$. This edge serves two purposes. First, its position defines a zero-bit position. Any following edges which overlap with this edge will be numerically interpreted as a zero-bit. Second, the position of this edge defines one unit along the x axis in the object's local coordinate system. The x axis for the object's local coordinate system thus starts at the local origin (i,j,k); one unit along the object's x axis extends to $(i+1,j,k)$.
4. Insert a one-edge polygon into the geometry going from (i,j,k) to $(i,j+1,k)$. This edge also serves two purposes. First, its position defines a one-bit position; any following edges that overlap with this edge will be numerically interpreted as a one-bit. Second, the position of this edge defines one unit along the y axis in the object's local coordinate system.
5. Insert a one-edge polygon into the geometry going from (i,j,k) to $(i,j,k+1)$. This edge serves only one purpose. The position of this edge defines one unit along the z axis in the object's local coordinate system. (Conceivably, we could also use this edge to make a ternary numerical encoding system, but alarmingly enough, some programmers these days even have problems with binary, so we'll just stick to binary.)
6. Insert two one-edge polygons into the geometry, forming an X on the tip of the x axis. This is for optical identification in Blender of which edge forms the x axis of the local coordinate system, since otherwise all axes would look alike.
7. We now wish to encode an integer identifier into the mesh. For each bit of the integer identifier, insert a one-edge polygon into the geometry. If the bit has a value of zero, insert a zero-bit edge; if it has a value of one, insert a one-bit edge. Continue until all bits of the integer identifier have been inserted.

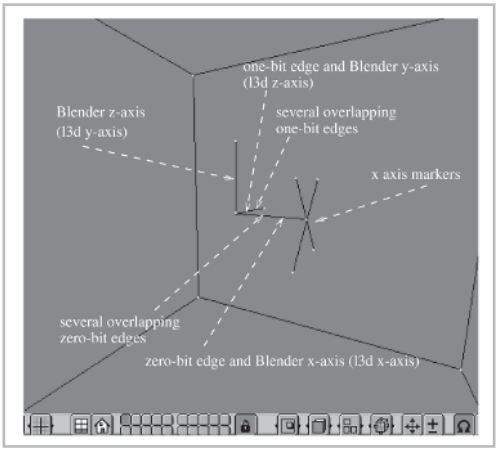


Figure 6-41: The appearance of the magic edges in Blender. Though it appears to be just composed of five edges, in reality several edges overlap to encode arbitrarily long integer numbers in a spatially small region. Notice that the Blender right-handed coordinate system has z going up and y into the page; l3d's left-handed coordinate system has y going up and z into the page.

The next question is exactly how these edges get inserted into the geometry. For a particular mesh, we can insert the magic edges by first exporting the mesh as Videoscape, parsing and augmenting the Videoscape file, deleting the original mesh in Blender, and then loading the augmented Videoscape file.

The problem with this approach is that it can be tedious to do all of these steps by hand in order to insert an ID into a mesh. Thus, a tool automates this process for us. This tool is `blend_at`, the topic of the next section.

The Tool Blend_at: Remote Control of Blender

The tool `blend_at` is a Blender attributes editor. When configured properly, it allows you to edit arbitrary attributes of any mesh object in Blender. The typical editing process using `blend_at` is as follows:

1. Select an object in Blender by right-clicking it.
2. Click **Refresh** in the `blend_at` window. The selected object's attributes are displayed.
3. Edit the object attributes, and click **Save** in the `blend_at` window.

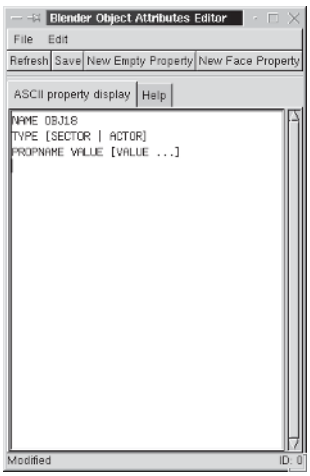


Figure 6-42: The main `blend_at` window.



CAUTION You must set up the default load and save filenames, described in detail below, before you can use the system in this manner.

`blend_at` is a program which runs parallel to Blender and automates Blender operations needed for our portal editing scheme. This allows for a fairly smooth, even if not perfectly integrated, editing of object attributes. The operation of the system is fairly simple, but it requires a bit of work to configure and understand the system. First, let's discuss how the system works, then look at the per-project configuration which needs to be done before using `blend_at`.

`blend_at` fulfills the following main goals:

- Automatic embedding of ID information into a mesh. We saw in the previous section how, technically, we use overlapping edges to store an integer ID within the mesh itself; the tedious part was the manual saving, changing, and reloading of the Vidoescape file. `blend_at` does this all automatically, by sending event messages via the X event mechanism to the Blender window, thereby effectively remote controlling the Blender window. The specific keystrokes that `blend_at` sends to Blender cause Blender to save the currently selected mesh to a Videoscape file. Then, `blend_at` scans the saved Videoscape file to see if an ID is already embedded by looking for overlapping one-edge polygons. If an ID is already found, we proceed to the next step. If no ID is found, then `blend_at` determines a new ID number by adding one to the last assigned ID number, which it keeps track of in an external file (typically `oid.txt`). `blend_at` then encodes the number in binary, inserts the appropriate one-edge polygons into the mesh, and sends Blender the appropriate keystrokes to cause it to delete the old mesh and reload the new mesh with the embedded ID.
- Association of attribute information with an ID. After finding the ID of the mesh, possibly creating a new ID if necessary, `blend_at` then looks for a filename `attribXXX.dat`, where `XXX` is the numerical ID of the selected mesh. The contents of this file are then displayed in the main text editing window of `blend_at`, where the file can be edited and saved. By convention, each line in the attributes file represents one attribute or property of the mesh. Each line starts with a keyword indicating the name of the property to set (such as `NAME`), followed by the value or values of the property. Therefore, all files with filenames `attribXXX.dat` form the attributes database, storing all extra information about all meshes. (The term database is used a bit loosely here, since we are using a collection of flat files, but we could just as easily use a real relational or object-oriented database to store the attribute information.)



NOTE The source and executable files for the program `blend_at` are located in directory `$L3D/source/utils/blendat/src`.

Configuration and Testing of blend_at

Now that we've seen what `blend_at` does (insertion of ID into mesh, association of attributes with ID) and how it does it (remote control of Blender through simulated keyboard events), let's look at its practical usage. Before you can use `blend_at`, you first must configure both Blender and `blend_at` to use the same directories and filenames. Do this as follows.

Team-Fly®

1. Start Blender and `blend_at`. It is typically most convenient to resize the Blender window to be slightly less wide, and to place the `blend_at` window next to the Blender window so that both can be seen simultaneously. In Blender, add any mesh object, such as a plane. Leave EditMode, and ensure that the plane object is selected.
2. Press **Alt+w**. Select a directory, and enter a filename ending in `.obj`. Say you choose the directory `/home/supergame` and filename `level.obj`. Confirm your choice by pressing **Enter** in the filename field. This saves the plane object to `/home/supergame/level.obj`. The purpose of this save operation is not to save the contents of a file, but rather to initialize Blender so that the next save operation automatically uses this same file. There is no reliable way for `blend_at` to instruct Blender to use a specific filename, so we must initialize a default filename in this way beforehand, and configure `blend_at` to use this known filename.
3. Press **F1**. Select the same directory you chose before, and the same filename you entered before. Confirm your choice by pressing **Enter** in the filename field. This loads the plane object from `/home/supergame/level.obj`. As before, the purpose of this load operation is not to load the contents of a file, but rather to initialize Blender so that the next load operation automatically uses this same file.
4. Type **x, Enter** to delete the just-loaded object, which is a second copy of the original mesh object added in step 1. Then, select the original mesh from step 1, and delete it as well; we no longer need it.
5. Check that Blender has accepted the default filenames you have entered, both for loading and saving. Do this as follows. Press **Alt+w** to check the default save filename. Blender should automatically display the directory and filename you entered earlier; in this example, `/home/supergame` and `level.obj`. Press **Esc** to cancel the save. Then, press **F1** to check the default load filename. Again, Blender should automatically display the same directory and filename. Press **Esc** to cancel the load. If either of these displays is incorrect, repeat the entire procedure again from step 1.
6. Configure `blend_at` to use the same directory and filename. Do this as follows. Start `blend_at`. In the menu, select **Edit** then **Preferences**. In the Blender working directory field enter the name of the directory from step 2. In the Blender workfile field enter the filename from step 2. In the Attribute database directory field enter the directory from step 2. In the OID filename field enter the string **oid.txt**—this is the file in which `blend_at` keeps track of the last ID number it assigned.

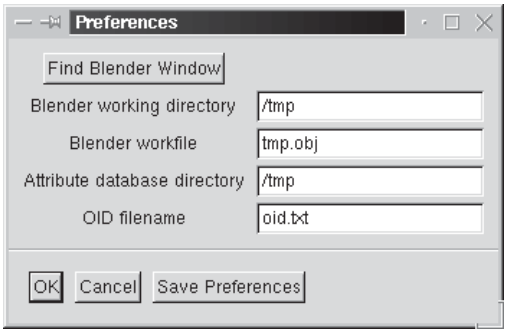


Figure 6-43: The configuration window for `blend_at`.

At this point, Blender and `blend_at` should be ready to use. Test your configuration as follows.



CAUTION Because `blend_at` sends simulated keystroke and mouse events to the Blender window, it is imperative that you not move the mouse or press any keys when `blend_at` is sending such events to Blender. The buttons Refresh and New Face Property cause the sending of events to Blender; thus, after clicking these buttons, do not move the mouse or press any keys until the command has completed (a few seconds).

1. Set up the Blender default load and save filenames as described above. Start `blend_at` and configure it to use the same directories and filenames.
2. Create two cube objects in Blender. Make sure to leave EditMode.
3. Select one of the cube objects. In `blend_at`, click **Refresh**. Wait until the command finishes executing. You should see the cursor move around automatically, and Blender should automatically reload the mesh. Notice that the mesh has a new set of magic edges, and that the `blend_at` window displays a new template attributes file for editing.
4. Type some random text into the `blend_at` text window. Click **Save**. This essentially saves this text with the mesh. (In actuality the text is associated with an ID number, and the ID number is encoded within the mesh, so the relationship between text and mesh is only indirect.)
5. Select the other cube object in Blender. In `blend_at`, click **Refresh** and wait for the command to complete. Notice the new magic edges, and that the `blend_at` window displays a new template attributes file for this second cube object.
6. Type some different random text into the `blend_at` window, and click **Save**.
7. Select the first cube object again in Blender. Click **Refresh** in `blend_at`. Notice that the text you originally typed for the first mesh appears again.

This example illustrates how the system effectively associates any arbitrary data with each mesh in Blender. Next, let's see specifically what data we should store with each mesh for the portalization system.

Specific Mesh Attributes Used by the Portalization System

The textual attribute data that we can associate with each mesh ultimately gets parsed by the `VidscParser.pm` module, where the attributes are then evaluated and affect the creation of the corresponding lines in the final world file. The following sections cover the attributes that are understood by `VidscParser.pm`, and how they affect the mesh with which they are associated.

The Name Attribute

Each attribute file should somewhere contain a line of the form:

```
NAME <MESH_NAME>
```

where `<MESH_NAME>` is a string name, without spaces, uniquely identifying the mesh. For instance, sectors might have names such as `living_room`, `bathroom`, or `the_zen_room`. Plug-in objects or actors might have names such as `torch5` or `amorphous_undulating_procedurally_generated_flesh_heap_number_69`. Typically, the name attribute appears as the first line in the file.

The Type Attribute

The attribute file should also somewhere contain a line of the form:

```
TYPE <TYPENAME>
```

where `<TYPENAME>` is the type of the mesh. Currently, types `SECTOR` and `ACTOR` are understood. Typically, the type attribute appears as the second line in the file.

The type of the mesh controls which other attributes in the attributes file are expected and understood. The next sections cover the attributes for mesh types `SECTOR` and `ACTOR` separately.

Attributes for Sectors

If the mesh is of type `SECTOR`, then the following additional attributes may appear in the world file to specify additional information about the sector.

FACE <X> Attributes

The `FACE` attribute specifies face-specific attributes about a mesh, and is used for texture mapping purposes. There are three allowable kinds of `FACE` attribute lines, covered in the following sections. All face attributes have in common that they begin with `FACE <X>`, where `<X>` represents the face number within the mesh. The faces of a mesh are stored in memory and exported to disk in a particular order; the *face number* is then a particular face's position in the list. `blend_at` can show you the face number for the selected faces in Blender. Do this as follows.

1. Enter EditMode.
2. Select the desired faces by right-clicking their vertices. If Draw Faces in the EditButtons is activated, each selected face is drawn in yellow. You can select multiple faces or just one.
3. Do not leave EditMode; you must remain in EditMode for the next step to work. (Note that this is in contrast to the procedure when using the Refresh button, in which case you must not be in EditMode.)

4. In `blend_at`, click **New Face Property**. Wait for the command to complete (a few seconds); do not move the mouse or press any keys until the command is done. For each selected face, `blend_at` creates a new `FACE` line in the text window with the appropriate numerical index.

The New Face Property button works by sending Blender keystrokes to cause it to duplicate and separate the selected faces, then save the separated faces to the work file. `blend_at` then reads in the face definitions from the file, sends Blender more keystrokes to cause it to save the original mesh, and then reads the original mesh into memory. Finally, `blend_at` searches for each selected face in the original mesh, and creates a corresponding `FACE` line with the numerical index in the original mesh of the found face.



CAUTION The New Face Property button assumes that the main geometry is located in Blender's layer 0, and uses layers 9 and 10 as temporary storage. Therefore, you should not have any geometry in layers 9 or 10, and you should normally work in layer 0.



CAUTION If you insert or delete faces into the sector mesh, the face numbers will very likely change, meaning that any `FACE` attribute specifications will now refer to different faces. Therefore, do not assign `FACE` attributes until the sector mesh geometry is fairly stable, or reassign `FACE` attributes after changing the sector mesh geometry.

FACE <X> IS_TEX_SPACE <NAME> <IMAGE>

The `FACE <X> IS_TEX_SPACE` attribute indicates that the particular face is a texture definition triangle, as we defined earlier in this chapter (in the discussion of the `Facet.pm` module). `<NAME>` is a string which identifies this texture space, including both its image and its orientation. The example we used earlier for a typical name was `stony_ceiling`; this represents the stony texture and the ceiling orientation. `<IMAGE>` is the name of the texture image to be used, such as `stone.ppm`.

The orientation of the texture space, as we saw earlier, is defined by the triangle itself. The triangle must be a right triangle. When looking at the front side of the triangle (the side into which the normal vector points; enable Draw Normals in Blender to see the normal vectors), the apex of the right angle is the origin of the texture space; the next point in the clockwise direction is one unit along the *u* axis, and the other point is one unit along the *v* axis.

Note that theoretically, textures—including image and orientation—can conceivably be shared among sectors, though in practice this is not very useful. For instance, we might want to assign the texture `stony_ceiling` to the ceiling polygons in several sectors. But this means that if we want another stony ceiling texture with a different orientation or position in another sector, we would need to define `stony_ceiling_2`, or `stony_ceiling_in_dungeon` and `stony_ceiling_in_hallway`. It turns out that it is more convenient to deal with textures—again, where the term refers to a combined image and orientation—on a per-sector basis. For this reason, internally, the `VidscParser.pm` module combines the sector and texture names to form a unique identifier, such as `dungeon.stony_ceiling` or `hallway.stony_ceiling`. Of course, the texture images themselves are only loaded once, since `World.pm` creates a unique list of all texture images used by all textures.

FACE <X> USES_TEX_SPACE <NAME>

The FACE <X> USES_TEX_SPACE attribute causes a polygon to use the texture space defined by a texture definition triangle. <NAME> is the name of the texture space to be used, and is implicitly understood to be sector-specific. This means that there must also be a texture definition triangle in the same sector that defines texture space <NAME>.

By assigning the same texture space to several polygons, all of the polygons will line up seamlessly on the edges. For instance, for a wall, you will typically define one texture space, then assign all polygons on the wall to use that same texture space. The next tutorial illustrates how to do this.

FACE <X> IS_INVISIBLE

The FACE <X> IS_INVISIBLE attribute only has a meaning for faces that are texture definition triangles. This attribute specifies that the face does not define any geometry (only a texture space) and should not appear within the final world file.

This implies that if we do not specify the IS_INVISIBLE attribute, a face can simultaneously be a texture definition triangle and a geometry polygon. This is useful if the mesh geometry already happens to contain a right triangle with exactly the right size and orientation; in this case, we can just reuse the existing geometry triangle as a texture definition triangle. Usually, though, such right triangles do not simply happen to exist in a mesh, and artificially introducing such triangles can be cumbersome. In these cases, which are more common, we then define extra triangles in the mesh without disrupting the existing geometry, and flag such extra triangles as being invisible texture definition triangles.

Attributes for Actors

If the mesh is of type ACTOR, then the following additional attributes may appear in the world file to specify additional information about the sector.

NAME <NAME>

The NAME attribute specifies a string without spaces which uniquely identifies the actor. <NAME> represents the actual string.

PARENT <SECTOR>

The PARENT attribute specifies the name of the parent sector, <SECTOR>, in which this object is initially located. The object is initially inserted into the objects list of the parent sector, so that it is drawn along with the sector.

Note that the parent sector does not own the object and is not responsible for its destruction; the object is merely located within the sector. Indeed, the object can move from sector to sector, in which case it must remove itself from the objects list of the old sector and insert itself into the objects list of the new sector.

IS_CAMERA YES

The IS_CAMERA attribute only appears if it has a value of YES. In the presence of such a line, the associated mesh is interpreted to be the camera object, and its starting position and parent sector

are used to initialize the camera location. In this case, all of the other actor attributes described below are currently ignored. (Though it would be possible to associate geometry with the camera object, as well.)

MESH <FILENAME.OBJ>

The `MESH` attribute specifies the name of the Videoscape mesh, `<FILENAME.OBJ>`, which defines the geometry of the plug-in object.

TEXTURE <FILENAME.PPM>

The `TEXTURE` attribute specifies the name of the texture image file, `<FILENAME.PPM>`, which should be applied to the geometry of the plug-in object.

TEXCOORDS <FILENAME.UV>

The `TEXCOORDS` attribute specifies the name of the texture coordinates file, `<FILENAME.UV>`, which should be used to map the texture file onto the polygons. See Chapter 3 for information on how to create this file.

PLUGIN <FILENAME.SO>

The `PLUGIN` attribute specifies the name of the dynamic library file, `<FILENAME.SO>`, which controls the behavior of this plug-in object. See the introductory companion book *Linux 3D Graphics Programming* or look at the included Makefiles on the CD-ROM for information on creating such dynamic library files.

Parsing of Attributes by VidscParser.pm and vidinfo

We have now seen how we use `blend_at` to associate attribute information with meshes, and what specific attributes have a meaning for the generation of the world file. This is all the write side of the attribute system, where we essentially insert information into the attributes database. The read side of the attribute system takes place in module `VidscParser.pm`, where the attribute information is extracted and evaluated. With an understanding of the operation of the attribute system, we now can examine those parts of `VidscParser.pm` that we skipped over in our earlier discussion.

The key to the entire attributes system is the unique identification of each mesh before and after Videoscape export, which we achieve via embedding of the binary encoding of an ID number through overlapping, specially positioned edges within the mesh geometry (a method which we can also refer to as smoke and mirrors). When parsing the Videoscape files, we need to extract the ID from the mesh geometry in order to find the corresponding attributes file `attribXXX.dat`. Extraction of the ID from the mesh geometry is done with the tool `vidinfo`.

Listing 6-12: `vidinfo.cc`

```
#include <unistd.h>
#include <stdio.h>
#include <vector>
#include <string>
```

```
struct vertex {
  float x,y,z;

  int operator==(const vertex &r) {
    return(
            -0.001 < x-r.x && x-r.x < 0.001 &&
            -0.001 < y-r.y && y-r.y < 0.001 &&
            -0.001 < z-r.z && z-r.z < 0.001
          );
  }
};

main(int argc, char **argv)
{

  if(argc != 2) {
    printf("usage: %s videoscape_file.obj", argv[0]);
    exit(-1);
  }
  FILE *fp;

  unsigned long oid;

  vector<vertex> vidscape_vertices;

  char vidscape_line[1024];

  float float1, float2, float3;
  int found=0;

  char blender_work_fname[1024];
  sprintf(blender_work_fname,argv[1]);
  fp = fopen(blender_work_fname, "rt");
  int lineno = 1;
  int num_vertices=0;

  float xsum=0.0, ysum=0.0, zsum=0.0;
  float xc,yc,zc;
  float x0_0,y0_0,z0_0  ,  x0_1,y0_1,z0_1,
  x1_0,y1_0,z1_0  ,  x1_1,y1_1,z1_1,
  x2_0,y2_0,z2_0  ,  x2_1,y2_1,z2_1;

  int prev_v_vtxcount=0, prev_v_vtx0_0, prev_v_vtx0_1;
  int overlapping_edge_count=0;

  vertex pos;
  vertex x_axis, y_axis, z_axis;

  while(!feof(fp) && !found) {
    fgets(vidscape_line, 1024, fp);
    if(!feof(fp)) {

#define VIDSC_VTXCOUNT_LINENO 2

      if(lineno>VIDSC_VTXCOUNT_LINENO &&
          lineno<=VIDSC_VTXCOUNT_LINENO + num_vertices)
      {
        vertex v;
        sscanf(vidscape_line, "%f %f %f", &float1,&float2,&float3);
```

```
  v.x =float1;
  v.y =float2;
  v.z =float3;
  vidscape_vertices.push_back(v);
}
if(lineno == VIDSC_VTXCOUNT_LINENO) {

  sscanf(vidscape_line, "%d", &num_vertices);
}

int sc;
int v_vtxcount=0, v_vtx0_0=0, v_vtx0_1=0;
unsigned long v_color;
char v_color_str[255];
if((lineno > VIDSC_VTXCOUNT_LINENO + num_vertices + 1)
    && (sc=sscanf(vidscape_line,"%d %d %d %s",
                  &v_vtxcount, &v_vtx0_0, &v_vtx0_1, v_color_str)==4)
    && (v_vtxcount==2)
  )
{

  if(prev_v_vtxcount==2
      &&((vidscape_vertices[v_vtx0_0]==vidscape_vertices[prev_v_vtx0_0]
          && vidscape_vertices[v_vtx0_1]==vidscape_vertices[prev_v_vtx0_1])
         ||(vidscape_vertices[v_vtx0_0]==vidscape_vertices[prev_v_vtx0_1]
            && vidscape_vertices[v_vtx0_1]==vidscape_vertices[prev_v_vtx0_0]))
    ) {
    overlapping_edge_count++;
  }else {
    overlapping_edge_count=1;
  }

  prev_v_vtxcount = v_vtxcount;
  prev_v_vtx0_0 = v_vtx0_0;
  prev_v_vtx0_1 = v_vtx0_1;
}

if(overlapping_edge_count==4)
{

  int x_v0, x_v1,
  y_v0, y_v1,
  z_v0, z_v1;

  fgets(vidscape_line, 1024, fp);
  sscanf(vidscape_line, "%d %d %d", &v_vtxcount, &x_v0, &x_v1);
  fgets(vidscape_line, 1024, fp);
  sscanf(vidscape_line, "%d %d %d", &v_vtxcount, &y_v0, &y_v1);
  fgets(vidscape_line, 1024, fp);
  sscanf(vidscape_line, "%d %d %d", &v_vtxcount, &z_v0, &z_v1);

  vertex x0,x1,y0,y1,z0,z1;
  if ( vidscape_vertices[x_v0] == vidscape_vertices[y_v0] ) {
    x0 = vidscape_vertices[x_v0];
    y0 = vidscape_vertices[y_v0];
    x1 = vidscape_vertices[x_v1];
    y1 = vidscape_vertices[y_v1];
  }else if(vidscape_vertices[x_v0] == vidscape_vertices[y_v1]) {
    x0 = vidscape_vertices[x_v0];
    y0 = vidscape_vertices[y_v1];
```

```
  x1 = vidscape_vertices[x_v1];
  y1 = vidscape_vertices[y_v0];
}else if(vidscape_vertices[x_v1] == vidscape_vertices[y_v0]) {
  x0 = vidscape_vertices[x_v1];
  y0 = vidscape_vertices[y_v0];
  x1 = vidscape_vertices[x_v0];
  y1 = vidscape_vertices[y_v1];
}else if(vidscape_vertices[x_v1] == vidscape_vertices[y_v1]) {
  x0 = vidscape_vertices[x_v1];
  y0 = vidscape_vertices[y_v1];
  x1 = vidscape_vertices[x_v0];
  y1 = vidscape_vertices[y_v0];
}else {
  fprintf(stderr,"fatal error: origin not found");
}

if ( vidscape_vertices[x_v0] == vidscape_vertices[z_v0] ) {
  z0 = vidscape_vertices[z_v0];
  z1 = vidscape_vertices[z_v1];
}else if(vidscape_vertices[x_v0] == vidscape_vertices[z_v1]) {
  z0 = vidscape_vertices[z_v1];
  z1 = vidscape_vertices[z_v0];
}else if(vidscape_vertices[x_v1] == vidscape_vertices[z_v0]) {
  z0 = vidscape_vertices[z_v0];
  z1 = vidscape_vertices[z_v1];
}else if(vidscape_vertices[x_v1] == vidscape_vertices[z_v1]) {
  z0 = vidscape_vertices[z_v1];
  z1 = vidscape_vertices[z_v0];
}else {
  fprintf(stderr,"fatal error: origin not found");
}

pos = x0;
x_axis.x = x1.x - x0.x;
x_axis.y = x1.y - x0.y;
x_axis.z = x1.z - x0.z;
y_axis.x = y1.x - x0.x;
y_axis.y = y1.y - x0.y;
y_axis.z = y1.z - x0.z;
z_axis.x = z1.x - x0.x;
z_axis.y = z1.y - x0.y;
z_axis.z = z1.z - x0.z;

int dummy1,dummy2,dummy3;
fgets(vidscape_line, 1024, fp);
sscanf(vidscape_line, "%d %d %d", &dummy1, &dummy2, &dummy3);
fgets(vidscape_line, 1024, fp);
sscanf(vidscape_line, "%d %d %d", &dummy1, &dummy2, &dummy3);

found = 1;
int done = 0;
int shift=0;
oid = 0;
while(!feof(fp) && !done) {
  fgets(vidscape_line, 1024, fp);
  if(!feof(fp)) {
    sscanf(vidscape_line,
           "%d %d %d",
           &v_vtxcount, &v_vtx0_0, &v_vtx0_1);
    if(v_vtxcount != 2) {
```

```
            done=1;
          }else {

            int bit = !
                      (((vidscape_vertices[v_vtx0_0].x==
                         vidscape_vertices[x_v0].x &&
                         vidscape_vertices[v_vtx0_0].y==
                         vidscape_vertices[x_v0].y &&
                         vidscape_vertices[v_vtx0_0].z==
                         vidscape_vertices[x_v0].z)
                        &&
                        (vidscape_vertices[v_vtx0_1].x==
                         vidscape_vertices[x_v1].x &&
                         vidscape_vertices[v_vtx0_1].y==
                         vidscape_vertices[x_v1].y &&
                         vidscape_vertices[v_vtx0_1].z==
                         vidscape_vertices[x_v1].z))
                       ||
                       ((vidscape_vertices[v_vtx0_0].x==
                         vidscape_vertices[x_v1].x &&
                         vidscape_vertices[v_vtx0_0].y==
                         vidscape_vertices[x_v1].y &&
                         vidscape_vertices[v_vtx0_0].z==
                         vidscape_vertices[x_v1].z)
                        &&
                        (vidscape_vertices[v_vtx0_1].x==
                         vidscape_vertices[x_v0].x &&
                         vidscape_vertices[v_vtx0_1].y==
                         vidscape_vertices[x_v0].y &&
                         vidscape_vertices[v_vtx0_1].z==
                         vidscape_vertices[x_v0].z)));
            oid |= bit<shift;
            shift++;
          }
        }
      }
      printf("%lu", oid);
    }

    lineno++;
  }
}
fclose(fp);

if(found) {
  printf("%f %f %f", pos.x, pos.y, pos.z);
  printf("%f %f %f", x_axis.x, y_axis.x, z_axis.x);
  printf("%f %f %f", x_axis.y, y_axis.y, z_axis.y);
  printf("%f %f %f", x_axis.z, y_axis.z, z_axis.z);
}
}
```

Tool `vidinfo` takes as a command-line parameter the name of a Videoscape file. It then produces as output either nothing, if no ID could be found within the mesh, or five lines (covered below) if an ID could be found within the mesh. The ID is found by scanning for the four overlapping one-edge polygons (start of transmission), then extracting the next three one-edge polygons as the zero-bit edge and *x* axis, the one-bit edge and *y* axis, and the *z* axis. We then scan the

Team-Fly®

remaining one-edge polygons and see if they overlap with the zero-bit edge or one-bit edge, and build up the ID number in binary fashion, until we run out of one-edge polygons.

Assuming `vidinfo` could find an ID within the file, then the five lines of output it produces are the ID number, the (*x*,*y*,*z*) location of the object in space, as determined by the intersection of the magic edges forming the local coordinate system, and three lines representing the object's rotation matrix, again determined by the local coordinate system formed by the magic edges. A typical output from `vidinfo` looks as follows.

```
19
-0.081439 2.115297 0.000000
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000
```

Note that the location and orientation are extracted directly from the Videoscape file and are thus specified in Blender's right-handed coordinate system, not l3d's left-handed coordinate system. VidscParser.pm changes the coordinates to the left-handed coordinate system before writing them to the world file.

`VidscParser.pm` uses `vidinfo` to extract the ID from the file. Then, with the ID, it tries to find the corresponding `attribXXX.dat` file, where `XXX` is the extracted mesh ID. The contents of the attributes file are read into an array in memory.

The `TYPE` and `NAME` attributes, since they are universal, are extracted first. Based on the type attribute, we either handle the mesh as an actor or as a sector.

If `TYPE` is `ACTOR`, then we extract all of the relevant actor attributes, such as plug-in name and texture file, which we need to create a valid `ACTOR` line as specified in Chapter 5. We store all of these attributes with the `Actor` object in memory. Later, during the generation phase, the `Actor` properties get written to the `ACTOR` line in the world file.

If `TYPE` is anything else, we assume that the mesh is a sector. (Ordinarily we would want to bring an error message for unknown types, but in this case, this technique allows completely untagged meshes to still be handled as unnamed sectors.) We already saw earlier how the vertices and polygons get created for a sector. What we didn't discuss earlier is that as each face is being parsed from Videoscape into the `World` structure, we search for any `FACE <X>` attribute in the mesh's attribute list. If we find any `FACE <X>` attributes corresponding to the current face being processed, then we use these attributes to control the exact attributes of the `Facet` object in memory, which in turn ultimately control what gets written to the world file. If the face is a visible texture definition triangle (`IS_TEX_SPACE`), then we compute the texture space based on the triangle and save the texture space by name in a hash for future reference by other polygons. If the face is an invisible texture triangle (`IS_TEX_SPACE` and `IS_INVISIBLE`), then we additionally set the face's flag `_is_invisible` to be true, which later inhibits the generation of this triangle as a geometry polygon in the world file. If the face uses a texture space (`USES_TEX_SPACE`), then the named texture is simply assigned to the facet. If the face uses no texture space (no special face attributes found), then no texture gets explicitly assigned to the face. Later, during the generation phase, faces with assigned textures cause the creation of `GEOMPOLY_TEX` lines with the texture space orientation and image specified by the assigned texture. Faces with no assigned textures also cause the creation of `GEOMPOLY_TEX` lines, but with a default texture space orientation, and with the first texture image available in the world file.

`VidscParser.pm` also assigns any actor objects to their parent sectors, as determined by the `PARENT` attribute for actor meshes. Then, during the generation phase, all actors belonging to a sector are printed immediately after the sector block, so that the actor gets assigned to the sector during loading of the world file.

Program Listings for blend_at

Now that we've seen all of the details of how `blend_at` works, we can look at the code listings for this tool.

`blend_at` was written using the GUI builder/environment Glade. The terms "builder/environment" are used to indicate that Glade is more than a one-way GUI builder tool, but not as completely integrated as a true Rapid Application Development environment. Glade is based upon the GTK toolkit of user interface widgets, and allows you to graphically place widgets within windows and attach code to the various events that can occur with widgets. Just as the world editing system frees us from worrying about the exact coordinates of objects in 3D space, Glade frees us from worrying about the exact coordinates of widgets within the window, and allows for much quicker, visual development of applications with modern user interfaces.

Glade automatically generates code to create the windows and place the widgets within the windows. This code was then manually extended to perform the necessary actions when a button is clicked or a menu item is selected. Rather than embedding all of the logic within the window classes themselves (which is a poor idea), the code follows the model-view-controller paradigm, where the view of data (the window) is kept separate from the actions taken upon it (the controller). The concept sounds simple enough in theory, but like all principles of good software design, the real issue is if you actually can stick to the concept in the code. In particular, with RAD environments and GUI builders, the temptation is great to just insert the application code in the empty function bodies conveniently provided by the GUI builder, but this approach scatters your application logic indiscriminately throughout the code, much as a poor BSP tree indiscriminately scatters coherent geometric detail into separate parts of the tree. The structure of complex application software should be decided by a competent designer; no GUI builder can suggest an appropriate software architecture for your application domain.

What this means is that the GUI code for `blend_at` is mostly automatically generated, and the parts which were extended (such as the actual code executed when a button is clicked) do not perform any of the logic themselves, but instead simply invoke methods on a controller object, which then carries out the actual task. The controller object thus encapsulates the logic. Therefore, we won't go over the GUI code for `blend_at` here, instead focusing on domain-specific classes in `blend_at`.

To distinguish the domain-specific files from the Glade-generated files, all domain-specific files begin with the prefix `nl_`, a prefix chosen for reasons printed on the front cover of the book. The next sections cover each of the domain-specific classes used in `blend_at`.

Class vertex

Class `vertex` is a vertex in 3D space. Member variables `x`, `y`, and `z` are the spatial coordinates. Operator `==` tests for equality of two vertex objects by comparing their coordinates, with a slight epsilon parameter to allow for nearby points to be counted as equal.

Listing 6-13: `nl_vertex.h`

```
#ifndef __NL_VERTEX_H
#define __NL_VERTEX_H

struct vertex {
  float x,y,z;

  int operator==(const vertex &r) {
    return(
            -0.001 < x-r.x && x-r.x < 0.001 &&
            -0.001 < y-r.y && y-r.y < 0.001 &&
            -0.001 < z-r.z && z-r.z < 0.001
          );
  }
};

#endif
```

Class blender_config

Class `blender_config` stores the Blender configuration data which `blend_at` must know in order to operate.

Member `blender_path` is a string storing the directory name where the Blender work file is located.

Member `blender_work_filename` is a string storing the filename of the Blender work file.

Member `db_path` is a string storing the directory in which the attribute files will be saved.

Member `oid_filename` is a string storing the filename in which `blend_at` stores a counter indicating the last ID number it assigned to a mesh. This counter is increased by one every time a mesh gets assigned a new ID, so that meshes always get unique ID numbers.

Constant `BLENDER_MAGIC_X_SIZE` is a floating-point value that determines the spatial size of the magic edges which get inserted into the geometry to encode the ID. By default, the size is 1.0.

Constant `ATTRIB_FNAME` is a string containing the prefix of all attribute filenames. By default, the prefix is the string `attrib`, leading to attribute filenames of the form `attribXXX.dat`.

Listing 6-14: `nl_blender_config.h`

```
#ifndef _BLENDER_CONFIG_H
#define _BLENDER_CONFIG_H

#include <string>

class blender_config {
  public:
    string blender_path;
```

```
    string blender_work_filename;
    string db_path;
    string oid_filename;

    static float BLENDER_MAGIC_X_SIZE;
    static char ATTRIB_FNAME[];
};

#endif
```

Listing 6-15: `nl_blender_config.cc`

```
#include "nl_blender_config.h"

float blender_config::BLENDER_MAGIC_X_SIZE = 1.0;
char blender_config::ATTRIB_FNAME[] = "attrib";
```

Class blender_controller

Class `blender_controller` is an abstract interface for remote control of Blender.

Member variable `blender_config` points to a Blender configuration object that stores important directory names and filenames.

Member variable `current_id` is an integer representing the ID of the mesh currently selected in Blender.

Member variable `current_prop_filename` is the full path to the current attributes file corresponding to the ID stored in `current_id`.

Member variable `property_lines` is a string list containing the contents of the current attributes file.

Member variable `selected_faces` is a list of integers indicating the indices of the faces which are currently selected in Blender.

Pure virtual method `find_blender_window` asks the controller to find the current Blender window and to somehow internally store a reference to this window, so that later commands can reference the Blender window. Exactly how the controller finds and stores a reference to the window depends on the windowing system, and is implemented in a subclass.

Pure virtual method `press_key` asks the controller to simulate a key-press event and send it to the Blender window.

Pure virtual method `release_key` asks the controller to simulate a key-release event and send it to the Blender window.

Pure virtual method `read_properties` asks the controller to extract the ID of the currently selected mesh in Blender, then to read the mesh's properties (the contents of its attributes files) into memory. If the currently selected mesh does not yet have an ID, then `read_properties` should create a new ID, embed the new ID in the mesh, reload the mesh, create an empty attributes file, and read this empty attributes file into memory.

Pure virtual method `determine_selected_face_nums` asks the controller to determine the indices of the faces currently selected in Blender, and to store this list in `selected_faces`. This list, as we saw earlier, is used by the New Face Property button to create new `FACE <X>` properties based on the currently selected faces.

Listing 6-16: nl_blender_controller.h

```
#ifndef __NL_BLENDER_CONTROLLER_H
#define __NL_BLENDER_CONTROLLER_H

#include <vector>

class blender_config;

class blender_controller {
  public:
    blender_config *config;

    virtual void find_blender_window(void) = 0;
    virtual void press_key(int code) = 0;
    virtual void release_key(int code) = 0;
    virtual void read_properties(void) = 0;
    virtual void determine_selected_face_nums(void) = 0;
    int current_id;
    char current_prop_filename[4096];
    vector<int> selected_faces;
    vector<string> property_lines;
};

#endif
```

Class blender_xcontroller

Class `blender_xcontroller` is a Blender controller for the X Window System.

Member variable `dpy` stores a pointer to the X Display.

Member variable `blender_w` stores a reference to the Blender window as a `Window` object.

Method `recursive_find_blender_window` recursively traverses all windows in the display until it finds a window with the title Blender. It then stores this window in the variable `blender_w`.

Overridden method `find_blender_window` calls `recursive_find_blender_window` to do the X search for the Blender window.

Overridden method `press_key` creates an event of type `XEvent`, fills the event structure so that it is a key-press event with the given key code, sends the event to the Blender window with `XSendEvent`, and then flushes the event queue with `XFlush` to ensure that the event is sent.

Overridden method `release_key` creates an event of type `XEvent`, fills the event structure so that it is a key-release event with the given key code, sends the event to the Blender window with `XSendEvent`, and then flushes the event queue with `XFlush` to ensure that the event is sent.

Overridden method `read_properties` is probably the most significant method in the entire `blend_at` program. This method sends keystrokes to Blender to cause it to save the currently selected mesh to the work file. Then, the file is read and scanned for an ID, by looking for overlapping edges as described earlier. If no ID is found, a new ID is created and encoded within the mesh; then, the appropriate keystrokes are sent to Blender to cause it to delete the old mesh and reload the new one with the embedded ID. After finding or newly assigning an ID, the attributes file corresponding to the ID is opened, and its contents are read into the `property_lines` list.

Overridden method `determine_selected_face_nums` is the second most significant method in `blend_at`. It determines which faces are selected by sending Blender keystrokes to separate and save the selection and the original mesh; then, it searches for the selected faces within the original mesh and stores their index numbers in the `selected_faces` list.

Listing 6-17: `nl_blender_xcontroller.h`

```
#ifndef __NL_BLENDER_XCONTROLLER_H
#define __NL_BLENDER_XCONTROLLER_H

#include <X11/X.h>
#include <X11/Intrinsic.h>
#include <unistd.h>
#include <stdio.h>
#include <gtk/gtk.h>
#include <vector>
#include <string>

#include "nl_blender_controller.h"

class blender_config;

class blender_xcontroller : public blender_controller {
  protected:
    Display *dpy;
    int recursive_find_blender_window(Window w);

  public:
    Window blender_w;

    blender_xcontroller(Display *dpy, blender_config *c);
    virtual ~blender_xcontroller(void);
    void find_blender_window(void);
    void press_key(int code);
    void release_key(int code);
    void read_properties(void);
    void determine_selected_face_nums(void);
};

#endif
```

Listing 6-18: `nl_blender_xcontroller.cc`

```
#include "nl_blender_xcontroller.h"
#include "nl_vertex.h"
#include "nl_blender_config.h"

blender_xcontroller::blender_xcontroller(Display *dpy, blender_config *c) {
  this->dpy = dpy;
  config = c;
  find_blender_window();
}

blender_xcontroller::~blender_xcontroller(void) {
  delete config;
}

int blender_xcontroller::recursive_find_blender_window(Window w) {
  Window root_return, parent_return, *children;
  unsigned int nchildren;
```

```
  if(blender_w) return 1;

  XQueryTree(dpy, w, &root_return, &parent_return,
             &children, &nchildren);
  if(children==NULL) {
    return 0;
  }else {
    int i;
    for(i=0; i<nchildren && !blender_w;  i++) {
      char *wn;
      XFetchName(dpy, children[i], &wn);
      if(wn && strcmp(wn, "Blender")==0) {
        blender_w = children[i];
      }else {
        recursive_find_blender_window(children[i]);
      }
      Xfree(wn);
    }
    Xfree(children);
  }
}

void blender_xcontroller::find_blender_window(void)
{
  blender_w = 0;
  recursive_find_blender_window(DefaultRootWindow(dpy));
}

void blender_xcontroller::press_key(int code)
{
  XEvent event;
  unsigned long mask;

  event.type = KeyPress;
  event.xkey.type = KeyPress;
  event.xkey.serial = 30;
  event.xkey.send_event = FALSE;
  event.xkey.display = dpy;
  event.xkey.window = blender_w;
  event.xkey.root = DefaultRootWindow(dpy);
  event.xkey.subwindow = 0;
  event.xkey.time =195454319O;
  event.xkey.x = event.xkey.y = 0;
  event.xkey.x_root = event.xkey.y_root = 0;
  event.xkey.state = 0;
  event.xkey.keycode = code;
  event.xkey.same_screen= TRUE;
  mask = 1;
  XSendEvent(dpy, blender_w,
             TRUE, mask, &event);

  Xflush(dpy);
}

void blender_xcontroller::release_key(int code)
{
  XEvent event;
  unsigned long mask;

  event.type = KeyRelease;
```

```
  event.xkey.type = KeyRelease;
  event.xkey.serial = 30;
  event.xkey.send_event = FALSE;
  event.xkey.display = dpy;
  event.xkey.window = (Window) blender_w;
  event.xkey.root = DefaultRootWindow(dpy);
  event.xkey.subwindow = 0;
  event.xkey.time =1954543190;
  event.xkey.x = event.xkey.y = 0;
  event.xkey.x_root = event.xkey.y_root = 0;
  event.xkey.state = 0;
  event.xkey.keycode = code;
  event.xkey.same_screen= TRUE;
  mask = 1;
  XSendEvent(dpy, blender_w,
             TRUE, mask, &event);

  Xflush(dpy);
}

void blender_xcontroller::read_properties(void)
{
  Window root_return, child_return;
  int rootx, rooty, winx, winy;
  int rootx_new, rooty_new, winx_new, winy_new;
  unsigned int mask;

  XQueryPointer(dpy, DefaultRootWindow(dpy), &root_return, &child_return,
                &rootx, &rooty, &winx, &winy, &mask);
  XWarpPointer(dpy, None, blender_w, 0, 0,
               0,0, 100,100);
  XQueryPointer(dpy, DefaultRootWindow(dpy), &root_return, &child_return,
                &rootx_new, &rooty_new, &winx_new, &winy_new, &mask);

  press_key(64);
  press_key(25);
  release_key(25);
  release_key(64);
  usleep(250000);
  press_key(36);
  usleep(250000);
  press_key(36);
  usleep(250000);
  press_key(36);
  usleep(250000);
  press_key(36);

  FILE *fp;

  unsigned long oid;

  vector<string> new_vidscape_strings;
  vector<vertex> vidscape_vertices;

  char oid_fname[1024];
  char attrib_fname[1024];
  sprintf(oid_fname,"%s/%s",
          config->db_path.c_str(),
          config->oid_filename.c_str());
```

```
  fp=fopen(oid_fname, "rt");
  char oid_txt[80];
  fgets(oid_txt, 80, fp);
  fclose(fp);
  sscanf(oid_txt, "%lu", &oid);
  oid++;

  int num_bits_in_oid=0;
  unsigned long temp_oid = oid;
  while(temp_oid) {
    num_bits_in_oid++;
    temp_oid>=1;
  }

  char vidscape_line[1024];
  string vidscape_string;
  float float1, float2, float3;
  int found=0;

  char blender_work_fname[1024];
  sprintf(blender_work_fname,"%s/%s",
          config->blender_path.c_str(),
          config->blender_work_filename.c_str());
  fp = fopen(blender_work_fname, "rt");
  int lineno = 1;
  int num_vertices=0;

  float xsum=0.0, ysum=0.0, zsum=0.0;
  float xc,yc,zc;
  float x0_0,y0_0,z0_0  ,  x0_1,y0_1,z0_1,
  x1_0,y1_0,z1_0  ,  x1_1,y1_1,z1_1,
  x2_0,y2_0,z2_0  ,  x2_1,y2_1,z2_1;

  int prev_v_vtxcount=0, prev_v_vtx0_0, prev_v_vtx0_1;
  int overlapping_edge_count=0;

  while(!feof(fp) && !found) {
    fgets(vidscape_line, 1024, fp);
    if(!feof(fp)) {

#define VIDSC_VTXCOUNT_LINENO 2

      if(lineno>VIDSC_VTXCOUNT_LINENO &&
          lineno<=VIDSC_VTXCOUNT_LINENO + num_vertices)
      {
        vertex v;
        sscanf(vidscape_line, "%f %f %f", &float1,&float2,&float3);
        xsum += float1;
        ysum += float2;
        zsum += float3;

        v.x =float1;
        v.y =float2;
        v.z =float3;
        vidscape_vertices.push_back(v);
      }

      if(lineno == VIDSC_VTXCOUNT_LINENO) {
```

```
  sscanf(vidscape_line, "%d", &num_vertices);

  sprintf(vidscape_line, "%d", num_vertices +
          (4 + 3 + 2 + num_bits_in_oid)*2);
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);
}else if (lineno == VIDSC_VTXCOUNT_LINENO + num_vertices) {
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);

  xc = xsum/num_vertices;
  yc = ysum/num_vertices;
  zc = zsum/num_vertices;

  x0_0 = xc; y0_0 = yc; z0_0 = zc;
  x0_1 = xc + blender_config::BLENDER_MAGIC_X_SIZE; y0_1 = yc; z0_1 = zc;
  x1_0 = xc; y1_0 = yc; z1_0 = zc;
  x1_1 = xc; y1_1 = yc + blender_config::BLENDER_MAGIC_X_SIZE; z1_1 = zc;
  x2_0 = xc; y2_0 = yc; z2_0 = zc;
  x2_1 = xc; y2_1 = yc; z2_1 = zc + blender_config::BLENDER_MAGIC_X_SIZE;

  for(int i_flagedge=0; i_flagedge<4; i_flagedge++) {
    sprintf(vidscape_line, "%f %f %f", x0_0, y0_0, z0_0);
    vidscape_string = vidscape_line;
    new_vidscape_strings.push_back(vidscape_string);
    sprintf(vidscape_line, "%f %f %f", x0_1, y0_1, z0_1);
    vidscape_string = vidscape_line;
    new_vidscape_strings.push_back(vidscape_string);
  }

  sprintf(vidscape_line, "%f %f %f", x0_0, y0_0, z0_0);
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);
  sprintf(vidscape_line, "%f %f %f", x0_1, y0_1, z0_1);
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);

  sprintf(vidscape_line, "%f %f %f", x1_0, y1_0, z1_0);
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);
  sprintf(vidscape_line, "%f %f %f", x1_1, y1_1, z1_1);
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);

  sprintf(vidscape_line, "%f %f %f", x2_0, y2_0, z2_0);
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);
  sprintf(vidscape_line, "%f %f %f", x2_1, y2_1, z2_1);
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);

  sprintf(vidscape_line, "%f %f %f", x0_1,
          y0_1 - 0.5,
          z0_1 - 0.5);
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);
  sprintf(vidscape_line, "%f %f %f", x0_1,
          y0_1 + 0.5,
          z0_1 + 0.5);
  vidscape_string = vidscape_line;
```

Team-Fly®

```
  new_vidscape_strings.push_back(vidscape_string);

  sprintf(vidscape_line, "%f %f %f", x0_1,
          y0_1 - 0.5,
          z0_1 + 0.5);
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);
  sprintf(vidscape_line, "%f %f %f", x0_1,
          y0_1 + 0.5,
          z0_1 - 0.5);
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);

  unsigned long temp_oid=oid;
  while(temp_oid) {
    if(temp_oid&0x1) {
      sprintf(vidscape_line, "%f %f %f", x1_0, y1_0, z1_0);
      vidscape_string = vidscape_line;
      new_vidscape_strings.push_back(vidscape_string);
      sprintf(vidscape_line, "%f %f %f", x1_1, y1_1, z1_1);
      vidscape_string = vidscape_line;
      new_vidscape_strings.push_back(vidscape_string);
    }
    else {
      sprintf(vidscape_line, "%f %f %f", x0_0, y0_0, z0_0);
      vidscape_string = vidscape_line;
      new_vidscape_strings.push_back(vidscape_string);
      sprintf(vidscape_line, "%f %f %f", x0_1, y0_1, z0_1);
      vidscape_string = vidscape_line;
      new_vidscape_strings.push_back(vidscape_string);
    }
    temp_oid>=1;
  }

}else {
  vidscape_string = vidscape_line;
  new_vidscape_strings.push_back(vidscape_string);
}

int sc;
int v_vtxcount, v_vtx0_0, v_vtx0_1;
unsigned long v_color;
char v_color_str[255];
if((lineno > VIDSC_VTXCOUNT_LINENO + num_vertices + 1)
    && (sc=sscanf(vidscape_line,"%d %d %d %s",
                  &v_vtxcount, &v_vtx0_0, &v_vtx0_1, v_color_str)==4)
    && (v_vtxcount==2)
  )
{

  if(prev_v_vtxcount==2
      &&((vidscape_vertices[v_vtx0_0]==vidscape_vertices[prev_v_vtx0_0]
          && vidscape_vertices[v_vtx0_1]==vidscape_vertices[prev_v_vtx0_1])
         ||(vidscape_vertices[v_vtx0_0]==vidscape_vertices[prev_v_vtx0_1]
           && vidscape_vertices[v_vtx0_1]==vidscape_vertices[prev_v_vtx0_0]))
    ) {
    overlapping_edge_count++;
  }else {
    overlapping_edge_count=1;
```

```
  }

  prev_v_vtxcount = v_vtxcount;
  prev_v_vtx0_0 = v_vtx0_0;
  prev_v_vtx0_1 = v_vtx0_1;
}

if(overlapping_edge_count==4)
{

  int x_v0, x_v1,
  y_v0, y_v1,
  z_v0, z_v1;

  fgets(vidscape_line, 1024, fp);
  sscanf(vidscape_line, "%d %d %d", &v_vtxcount, &x_v0, &x_v1);
  fgets(vidscape_line, 1024, fp);
  sscanf(vidscape_line, "%d %d %d", &v_vtxcount, &y_v0, &y_v1);
  fgets(vidscape_line, 1024, fp);
  sscanf(vidscape_line, "%d %d %d", &v_vtxcount, &z_v0, &z_v1);

  int dummy1,dummy2,dummy3;
  fgets(vidscape_line, 1024, fp);
  sscanf(vidscape_line, "%d %d %d", &dummy1, &dummy2, &dummy3);
  fgets(vidscape_line, 1024, fp);
  sscanf(vidscape_line, "%d %d %d", &dummy1, &dummy2, &dummy3);

  found = 1;
  int done = 0;
  int shift=0;
  oid = 0;
  while(!feof(fp) && !done) {
    fgets(vidscape_line, 1024, fp);
    if(!feof(fp)) {
      sscanf(vidscape_line,
             "%d %d %d",
             &v_vtxcount, &v_vtx0_0, &v_vtx0_1);
      if(v_vtxcount != 2) {
        done=1;
      }else {

        int bit = !
                  (((vidscape_vertices[v_vtx0_0].x==
                     vidscape_vertices[x_v0].x &&
                     vidscape_vertices[v_vtx0_0].y==
                     vidscape_vertices[x_v0].y &&
                     vidscape_vertices[v_vtx0_0].z==
                     vidscape_vertices[x_v0].z)
                    &&
                    (vidscape_vertices[v_vtx0_1].x==
                     vidscape_vertices[x_v1].x &&
                     vidscape_vertices[v_vtx0_1].y==
                     vidscape_vertices[x_v1].y &&
                     vidscape_vertices[v_vtx0_1].z==
                     vidscape_vertices[x_v1].z))
                   ||
                   ((vidscape_vertices[v_vtx0_0].x==
                     vidscape_vertices[x_v1].x &&
                     vidscape_vertices[v_vtx0_0].y==
                     vidscape_vertices[x_v1].y &&
```

```
                        vidscape_vertices[v_vtx0_0].z==
                        vidscape_vertices[x_v1].z)
                       &&
                       (vidscape_vertices[v_vtx0_1].x==
                        vidscape_vertices[x_v0].x &&
                        vidscape_vertices[v_vtx0_1].y==
                        vidscape_vertices[x_v0].y &&
                        vidscape_vertices[v_vtx0_1].z==
                        vidscape_vertices[x_v0].z)));
            oid |= bit<shift;
            shift++;
          }
        }
      }
      sprintf(attrib_fname, "%s/%s%lu.dat",
              config->db_path.c_str(),
              blender_config::ATTRIB_FNAME,
              oid);
      this->current_id = oid;
    }

    lineno++;
  }
}
fclose(fp);

if(!found) {

  int magic_vtx_ctr;

  for(magic_vtx_ctr=0; magic_vtx_ctr < 4*2; magic_vtx_ctr+=2) {

    sprintf(vidscape_line, "2 %d %d 0x%x",
            num_vertices+magic_vtx_ctr, num_vertices+magic_vtx_ctr+1,
            0x0);
    vidscape_string=vidscape_line;
    new_vidscape_strings.push_back(vidscape_line);
  }

  int stop=magic_vtx_ctr + 3*2;
  for(; magic_vtx_ctr < stop; magic_vtx_ctr+=2) {

    sprintf(vidscape_line, "2 %d %d 0x%x",
            num_vertices+magic_vtx_ctr, num_vertices+magic_vtx_ctr+1,
            0x0);
    vidscape_string=vidscape_line;
    new_vidscape_strings.push_back(vidscape_line);
  }

  stop=magic_vtx_ctr + 2*2;
  for(; magic_vtx_ctr < stop; magic_vtx_ctr+=2) {
    sprintf(vidscape_line, "2 %d %d 0x%x",
            num_vertices+magic_vtx_ctr, num_vertices+magic_vtx_ctr+1,
            0x0);
    vidscape_string=vidscape_line;
    new_vidscape_strings.push_back(vidscape_line);
  }

  unsigned long temp_oid = oid;
  while(temp_oid) {
```

```
    sprintf(vidscape_line, "2 %d %d 0x%x",
            num_vertices+magic_vtx_ctr,
            num_vertices+magic_vtx_ctr+1,
            0x0);
    magic_vtx_ctr+=2;
    vidscape_string=vidscape_line;
    new_vidscape_strings.push_back(vidscape_line);
    temp_oid >= 1;
  }

  fp=fopen(oid_fname, "wt");
  fprintf(fp, "%d", oid);
  fclose(fp);

  fp=fopen(blender_work_fname, "wt");
  for(int i=0; i<new_vidscape_strings.size(); i++) {
    fprintf(fp,"%s", new_vidscape_strings[i].c_str());
  }
  fclose(fp);

  for(int i=0; i<new_vidscape_strings.size(); i++) {
    new_vidscape_strings.pop_back();
  }

  press_key(53);
  release_key(53);
  usleep(250000);
  press_key(36);

  press_key(67);
  release_key(67);
  usleep(250000);
  press_key(36);
  usleep(250000);
  press_key(36);
  usleep(250000);
  press_key(36);
  usleep(250000);
  press_key(36);

  sprintf(attrib_fname, "%s/%s%lu.dat",
          config->db_path.c_str(),
          config->ATTRIB_FNAME,
          oid);
  fp = fopen(attrib_fname, "wt");
  fprintf(fp, "NAME OBJ%d", oid);
  fprintf(fp, "TYPE [SECTOR | ACTOR]");
  fprintf(fp, "PROPNAME VALUE [VALUE ...]");
  fclose(fp);

}

XWarpPointer(dpy, None, None, 0, 0,
             0,0, rootx - rootx_new,rooty-rooty_new);
Xflush(dpy);

int old_linecount = property_lines.size();
int i;
for(i=0; i<old_linecount; i++) {property_lines.pop_back(); }
```

```
  strncpy(current_prop_filename, attrib_fname, sizeof(current_prop_filename));
  fp = fopen(attrib_fname, "rt");

  if(fp) {
    char attrib_line[4096];
    while(!feof(fp)) {
      fgets(attrib_line, sizeof(attrib_line), fp);
      if(feof(fp)) break;
      property_lines.push_back(attrib_line);
    }
    fclose(fp);
  }else {
    printf("error opening %s", attrib_fname);
  }
}

void blender_xcontroller::determine_selected_face_nums(void)
{
  int i;

  int oldsize = this->selected_faces.size();
  for(i=0; i<oldsize; i++) {
    this->selected_faces.pop_back();
  }

  Window root_return, child_return;
  int rootx, rooty, winx, winy;
  int rootx_new, rooty_new, winx_new, winy_new;
  unsigned int mask;

  XQueryPointer(dpy, DefaultRootWindow(dpy), &root_return, &child_return,
                &rootx, &rooty, &winx, &winy, &mask);
  XWarpPointer(dpy, None, blender_w, 0, 0,
               0,0, 100,100);
  XQueryPointer(dpy, DefaultRootWindow(dpy), &root_return, &child_return,
                &rootx_new, &rooty_new, &winx_new, &winy_new, &mask);

  press_key(58);
  release_key(58);
  usleep(250000);
  press_key(19);
  usleep(250000);
  release_key(19);
  usleep(250000);
  press_key(36);
  usleep(250000);

  press_key(19);
  usleep(250000);
  release_key(19);
  usleep(250000);

  press_key(50);
  usleep(250000);
  press_key(40);
  usleep(250000);
  release_key(50);
  release_key(40);
  press_key(36);
  usleep(250000);
```

```
press_key(33);
usleep(250000);
release_key(33);
press_key(36);
usleep(250000);
press_key(58);
usleep(250000);
press_key(18);
release_key(18);
usleep(250000);
press_key(36);
usleep(250000);

press_key(23);
press_key(38);
press_key(64);
usleep(250000);
press_key(25);
usleep(250000);
release_key(25);
release_key(64);
usleep(250000);
press_key(36);
usleep(250000);
press_key(36);

usleep(250000);
usleep(250000);
usleep(250000);
usleep(250000);

press_key(53);
usleep(250000);
press_key(36);

int num_face_vertices;
vector<vertex> face_vertices;
typedef vector<int> facedef;
vector<facedef> selected_faces;
int num_selected_faces;
num_selected_faces = 0;

FILE *fp;
char blender_work_fname[1024];
char line[4096];
sprintf(blender_work_fname,"%s/%s",
        config->blender_path.c_str(),
        config->blender_work_filename.c_str());
fp = fopen(blender_work_fname, "rt");
fgets(line, sizeof(line), fp);
fgets(line, sizeof(line), fp);
sscanf(line, "%d", &num_face_vertices);

for(i=0; i<num_face_vertices; i++) {
  vertex v;
  fgets(line, sizeof(line), fp);
  sscanf(line, "%f %f %f",
         &v.x,
         &v.y,
         &v.z);
```

```
  face_vertices.push_back(v);
}
while(!feof(fp)) {
  fgets(line, sizeof(line), fp);
  if(feof(fp)) break;

  char *tok;
  tok = strtok(line," ");
  int numv;
  sscanf(tok, "%d", &numv);

  facedef a_facedef;
  a_facedef.push_back(numv);
  int iv;
  for(iv=0; iv<numv; iv++) {
    tok = strtok(NULL, " ");
    int ivertex;
    sscanf(tok, "%d", &ivertex);
    a_facedef.push_back(ivertex);
  }
  selected_faces.push_back(a_facedef);
  num_selected_faces++;
}
fclose(fp);

press_key(18);
release_key(18);
usleep(250000);

press_key(38);
usleep(250000);
press_key(38);
usleep(250000);
press_key(64);
usleep(250000);
press_key(25);
usleep(250000);
release_key(25);
release_key(64);
usleep(250000);
press_key(36);
usleep(250000);
press_key(36);

press_key(58);
release_key(58);
usleep(250000);
press_key(10);
usleep(250000);
release_key(10);
usleep(250000);
press_key(36);
usleep(250000);

press_key(10);
release_key(18);
usleep(250000);

XWarpPointer(dpy, None, None, 0, 0,
             0,0, rootx - rootx_new,rooty-rooty_new);
```

```
Xflush(dpy);

usleep(250000);
usleep(250000);
usleep(250000);
usleep(250000);

int num_mesh_vertices;
vector<vertex> mesh_vertices;
sprintf(blender_work_fname,"%s/%s",
        config->blender_path.c_str(),
        config->blender_work_filename.c_str());
fp = fopen(blender_work_fname, "rt");
fgets(line, sizeof(line), fp);
fgets(line, sizeof(line), fp);
sscanf(line, "%d", &num_mesh_vertices);
for(i=0; i<num_mesh_vertices; i++) {
  vertex v;
  fgets(line, sizeof(line), fp);
  sscanf(line, "%f %f %f",
         &v.x,
         &v.y,
         &v.z);
  mesh_vertices.push_back(v);
}

int found_overlapping_face;
int face_num;
face_num = 0;
while(!feof(fp)) {
  fgets(line, sizeof(line), fp);
  if(feof(fp)) break;

  int selface;
  char orig_line[4096];
  strncpy(orig_line, line, sizeof(orig_line));
  for(selface=0; selface<num_selected_faces; selface++) {
    strncpy(line, orig_line, sizeof(line));

    char *tok;
    tok = strtok(line, " ");

    int num_v;
    int iv;
    sscanf(tok, "%d", &num_v);

    int found_overlapping_vertex;
    for(iv=0; iv<num_v; iv++) {
      tok = strtok(NULL, " ");
      int ivertex;
      sscanf(tok, "%d", &ivertex);

      found_overlapping_vertex = 0;
      int fv;
      for(fv=1; fv<=selected_faces[selface][0]; fv++) {
        if ( mesh_vertices[ivertex] ==
             face_vertices[selected_faces[selface][fv]] )
        {
          found_overlapping_vertex = 1;
          break;
```

```
          }
        }
        if(!found_overlapping_vertex) {
          break;
        }
      }
      if(found_overlapping_vertex) {
        this->selected_faces.push_back(face_num);
        break;
      }
    }
    face_num++;
  }
  fclose(fp);
}
```

Tutorial: Creating a Textured Room with Actors

In this last tutorial, we now use all of the features of `blend_at` to create a textured room with plug-in objects. For simplicity, we'll just create one room, rather than creating several rooms. The earlier tutorials in this chapter already showed you how to create several rooms connected with portals; here, the focus is on the new specifications of texture and actor information. The files from this example may be found in directory `$L3D/data/levels/bigcube`.

1. Start Blender and `blend_at`. Choose a directory and filename for the work file, and configure both Blender and `blend_at` to use this directory and filename.
2. Create a fairly large cube in Blender. Subdivide the entire cube twice so that each side has 16 faces. Ensure that the normals point inward (**Shift+Ctrl+n**). This cube will be our sector. Then, exit EditMode. Ensure that the cube object is selected.

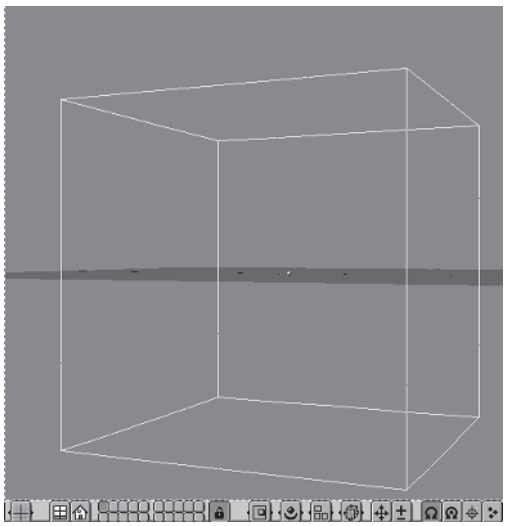


Figure 6-44

3. In `blend_at`, click **Refresh** and wait for the command to complete. Notice that the Blender window acts under the control of the `blend_at` program. When the command is complete,

`blend_at` displays a new, empty attributes file, and the mesh in Blender now contains an embedded ID.

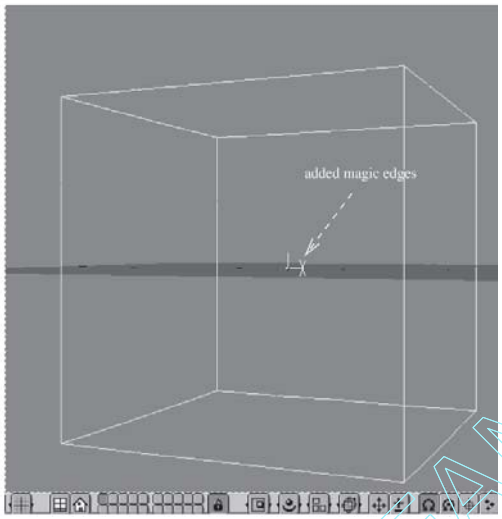


Figure 6-45

TEAMFLY

4. In `blend_at`, enter the following attributes.

```
TYPE SECTOR
NAME CUBE1
```

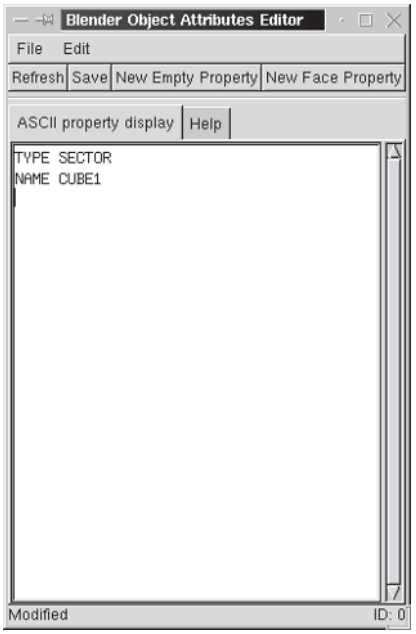


Figure 6-46

5. Create six new texture definition triangles, one slightly outside of each side of the cube. There are several ways to do this, but I recommend the following procedure.

Team-Fly®

5a. Enter EditMode. Select a face which has approximately the same size as the desired texture definition triangle.

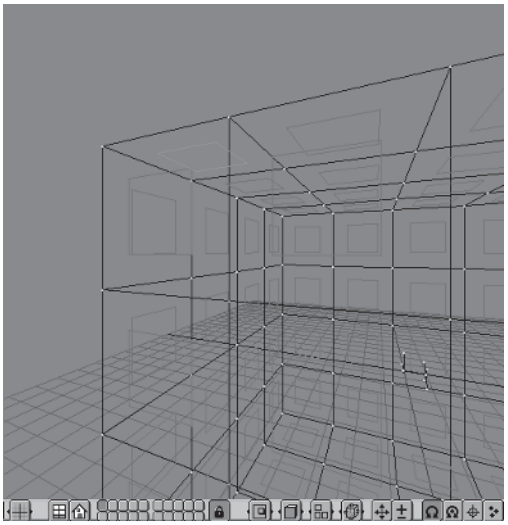


Figure 6-47

5b. Switch to an orthogonal view so that you are now viewing the selected face on its edge. For instance, if you selected a face on the top of the cube, you would now change to front or side orthogonal view; if you selected a face on the right side of the cube, you would change to front or top orthogonal view.

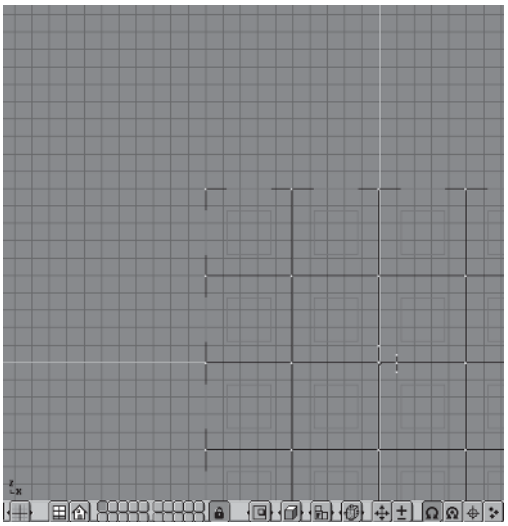


Figure 6-48

5c. Press **Shift+d** to duplicate the selected face. Press one of the arrow keys to nudge the face slightly so that it is located just slightly outside of the cube, then press **Enter**. At this point, you have just created a face parallel to the side of the cube from which it was duplicated, but located slightly away from the cube.

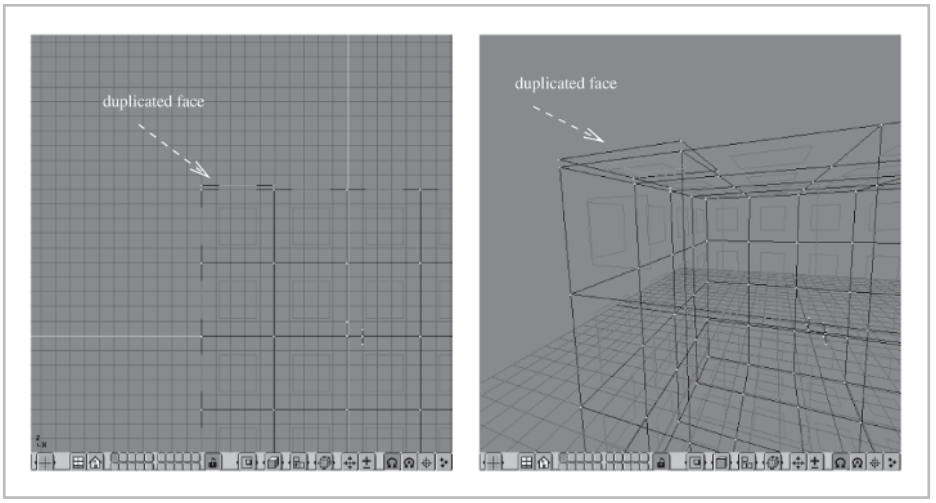


Figure 6-49

5d. Select all vertices except for the just duplicated face, and type **h** to hide them.

5e. Ensure that the face is a right triangle. Remember, texture definition triangles must be right triangles. If the face is not a triangle but is instead a quad (four vertices), then delete one of its vertices by right-clicking the vertex and pressing **x**. Then, select all three remaining vertices and press **f** to make a triangular face. After ensuring that the face is triangular, snap all the vertices to the nearest grid point (select the vertices, press **Shift+s**, then select **Sel->Grid**), then drag the vertices, if necessary, to make the triangle exactly a right triangle.

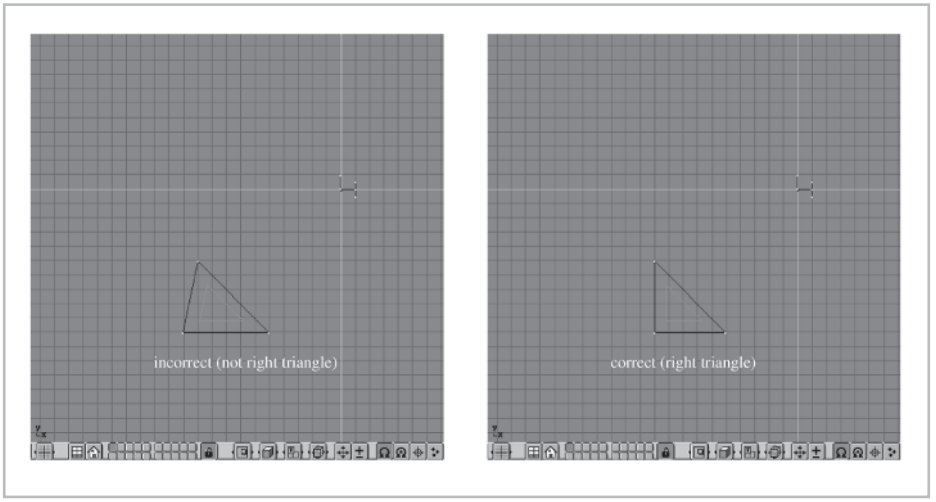


Figure 6-50

5f. Press **Alt+h** to unhide the vertices. Ensure that the normal vector for the face points towards the inside of the cube. If it does not, flip it with **Shift+Ctrl+n** or **Ctrl+n**.

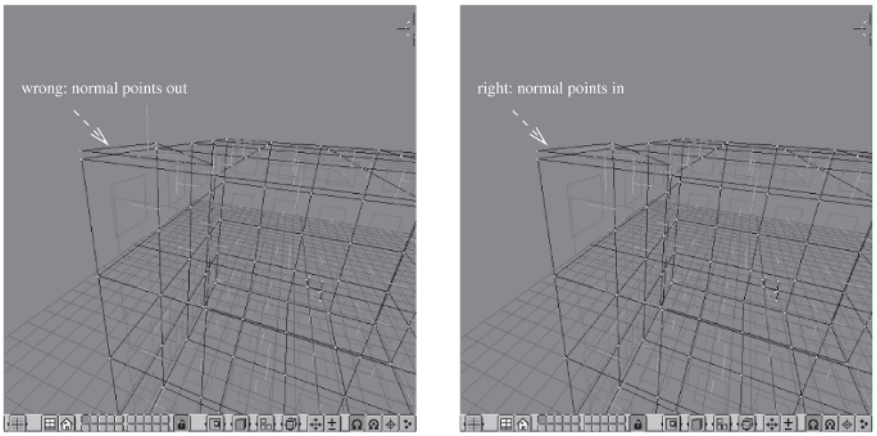


Figure 6-51

5g. Repeat this process from step 5a for each of the remaining five sides of the cube.

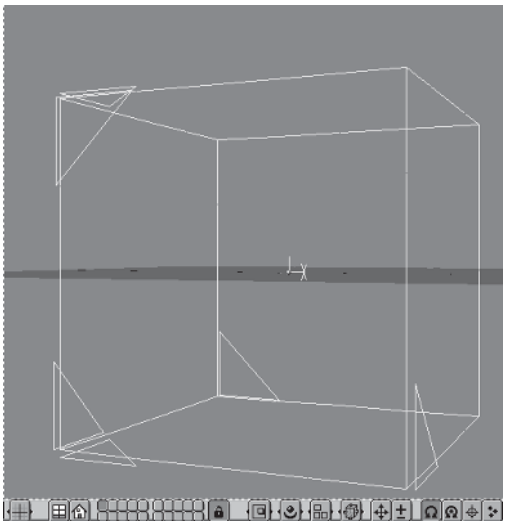


Figure 6-52: Shown for clarity outside of EditMode, though in the tutorial you should still be in EditMode.

6. Mark each texture definition triangle as such by using `blend_at`. Select the triangle and remain in EditMode. Click New Face Property in `blend_at` and wait for the command to complete. Notice the new `FACE <X>` line which appears in the text window, where `<X>` is the automatically determined face number.

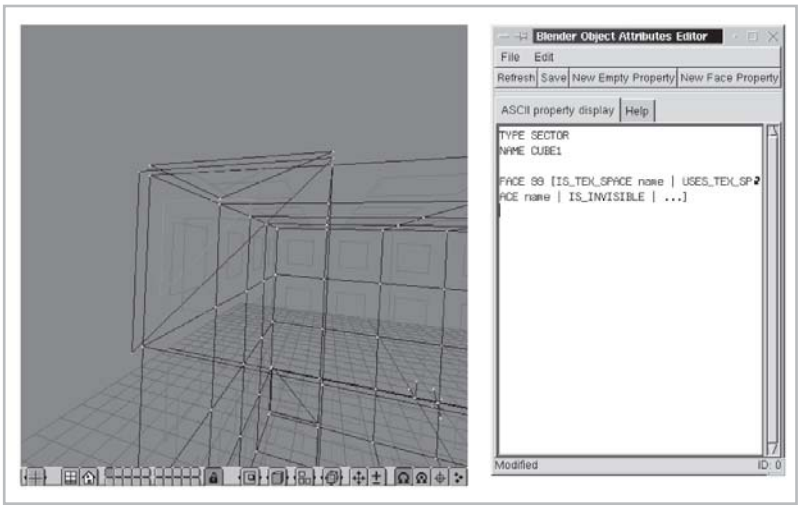


Figure 6-53

7. Edit the line to read

   ```
   FACE <X> IS_TEX_SPACE <NAME> <IMAGE>
   ```

 For <NAME>, substitute an appropriate name for the texture space. In this case, we have exactly six texture spaces, one for each side of the cube. Logical names for these are FRONT, BACK, LEFT, RIGHT, TOP, and BOTTOM. I use the convention that when viewing the model in front view (press **1** on the numeric keypad), the apparent left and right sides are LEFT and RIGHT sides; from side view (press **3** on the numeric keypad), the apparent left and right sides are the FRONT and BACK sides, and the apparent top and bottom sides are the TOP and BOTTOM sides. For <IMAGE>, substitute the string `stone.ppm`, or any other valid texture file.

8. Add another line directly beneath this one reading:

   ```
   FACE <X> IS_INVISIBLE
   ```

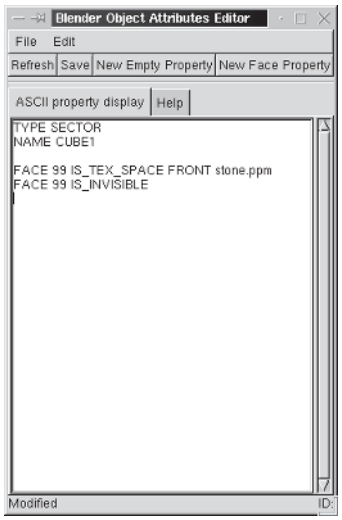


Figure 6-54

9. Deselect all vertices, then select all faces which should use this texture space. For instance, if you just defined the FRONT texture space, then select all faces on the top of the cube.

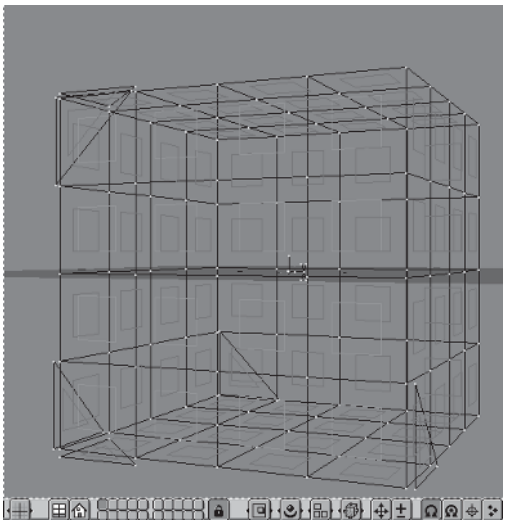


Figure 6-55

10. Click **New Face Property** in `blend_at` and wait for the command to complete. Notice that `blend_at` creates several new `FACE <X>` lines, one for each selected face. Edit each of these lines to read:

    ```
    FACE <X> USES_TEX_SPACE <NAME>
    ```

 where `<NAME>` is the name of the previously defined texture space, such as FRONT. Since there will be several faces using the same texture space, you can copy and paste text between lines to save work. Press **Ctrl+c** and **Ctrl+v** to copy and paste text, respectively.
11. Repeat this process of face classification by repeating from step 6 until all faces have been classified.
12. Click **Save** to save your changes to the attributes file.

Let's step back for a moment and summarize what we have just done. We have created six extra right triangles, all located parallel to but slightly outside of the cube. We then marked each of these triangles as being a texture space, and then assigned the faces from each side of the cube to use the appropriate one of the six texture spaces.

One item deserves special comment, the reason that we create the triangles slightly outside of the cube. We do this for two reasons. First of all, we can easily see that these triangles are separate from the mesh geometry, giving us a visual cue that they are texture definition triangles. Second, the fact that the texture definition triangles are topologically separate from the rest of the mesh allows us to use a convenient feature of Blender, *linked selection*. Linked selection works in EditMode, and selects vertices starting at the mouse cursor location, extending the selection to all vertices that are connected by an edge to the original vertex. To try this, select the cube and enter EditMode. Move the mouse cursor near the middle of the cube, next to some vertex in the middle, but away from the magic vertices. Then press **l** for linked select. Notice that just the cube has been

selected, and none of the texture definition triangles. Next, press **h**. This hides the cube and conveniently leaves just the texture definition triangles visible, allowing for easy editing.

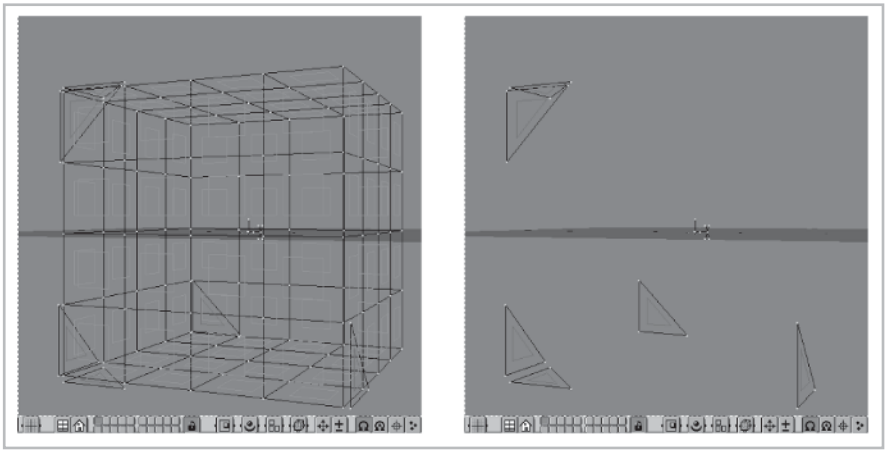


Figure 6-56: With two keystrokes we can select and hide the mesh geometry, leaving the texture definition triangles intact.

At this point in this tutorial, we've created a textured cube, but have not yet put in any plug-in objects. Let's now create a plug-in object and a camera object.

1. Exit EditMode. Add a new, separate cube object, sized somewhat small and located somewhere inside the larger cube. After adding the cube, exit EditMode, and ensure that the smaller cube is still selected.

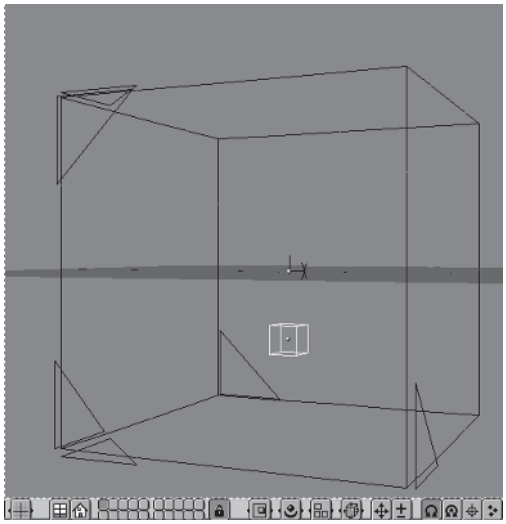


Figure 6-57

2. In `blend_at`, click **Refresh** and wait for the command to complete. Notice that `blend_at` inserts magic edges into the mesh and displays a new empty attributes file.
3. Enter the following attributes into the attributes file. Click **Save** when you are done.

```
NAME TORCH1
TYPE ACTOR
PARENT CUBE1
```

```
MESH torch.obj
TEXCOORDS torch.uv
TEXTURE torch.ppm
PLUGIN ../lib/dynamics/plugins/vidmesh/vidmesh.so
```

Note that all of the files specified in the attributes exist in the executable directory for the `porlotex` program of Chapter 5, which we have been using to test our world files. In general, any files you specify should be accessible from the directory containing the world file and the executable program file. As you can see from the `PLUGIN` attribute, it is permitted to specify relative pathnames.

4. In Blender, rotate the cube so that its local coordinate system tilts slightly backwards towards the front of the cube. Do this rotation outside of EditMode, so that you can easily undo it later (with **Alt+r**) if you wish.

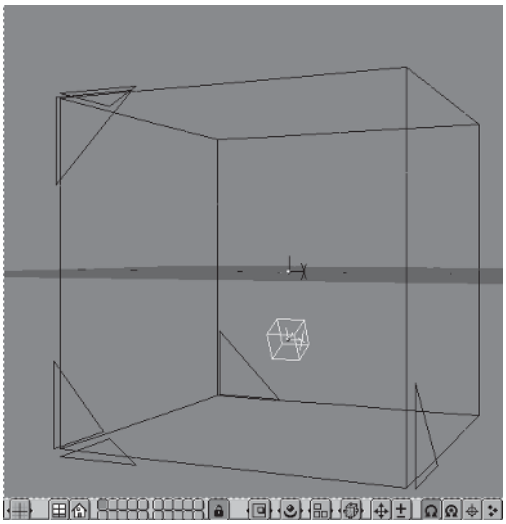


Figure 6-58

5. Place a camera in the scene as follows: Ensure you are not in EditMode. Insert a second cube somewhere inside the sector, exit EditMode, and ensure the second cube is selected. Click **Refresh** in `blend_at`, wait for the command to complete, and enter the following attributes. Click **Save** when you are done.

```
NAME CAMERA1
TYPE ACTOR
PARENT CUBE1
IS_CAMERA yes
PLUGIN no_plugin.so
```

Remember that the `PLUGIN` attribute is currently ignored for camera objects, but we still need to specify it.

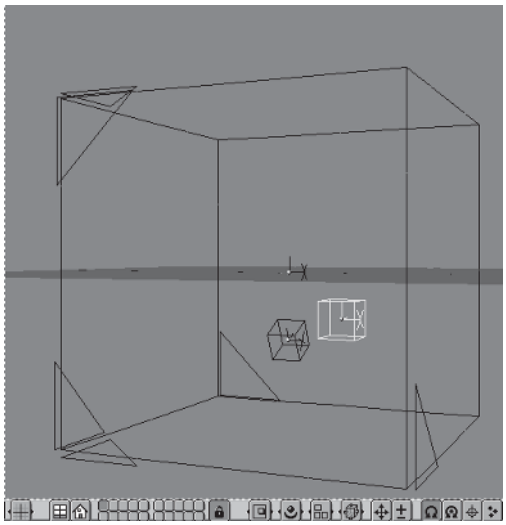


Figure 6-59

At this point, we are ready to export, convert, and test the level. Do this as follows:

1. Mark all meshes in Blender with **Ctrl+a**. Export them all to Videoscape by pressing **Alt+w, Enter, Enter**. Since you already set up the default filename earlier, you don't need to explicitly specify it here.
2. Run the Videoscape conversion process by entering **perl $L3D/source/utils/portalize/vid2port.pl <PATH>/mesh.obj* > world.dat**, where `<PATH>` is the directory where you saved the Videoscape files and `mesh.obj` is the filename you used. Open the world file in a text editor, and notice that the texture list at the beginning of the file has been automatically generated, as have the `ACTOR` and `CAMERA` lines with the proper coordinates for the actors and camera.
3. Copy the resulting `world.dat` file into the same directory as the executable file `porlotex`. Ensure that all files referenced within `world.dat` (texture, mesh, and plug-in files) are in the current directory. If you used the filenames provided above, the files will already be in the `porlotex` executable directory.
4. Run the program `porlotex`. Notice that the camera's starting position is as specified in the world file, and that the plug-in object, which was a cube in Blender, has been replaced with the specified mesh geometry, in this case a torch. Also notice that the torch is properly tilted backwards, just as we oriented it in Blender. Finally, notice that the textures on the walls are seamlessly tiled, because all polygons for a wall share the same texture space. Even if the polygons were oddly triangulated and of different sizes and orientations, as is often the case when faces are subdivided to cut holes into sectors, the textures still line up perfectly because of the shared texture space.

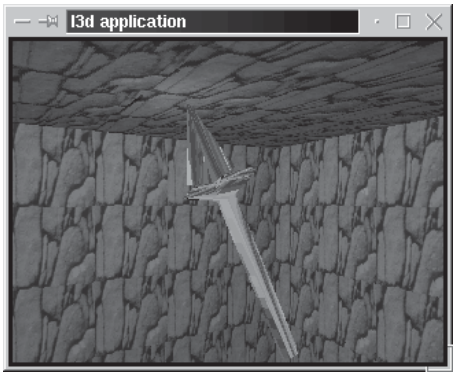


Figure 6-60: Output from program `porlotex` with the new world file created via the world editing system. A lamp has been manually placed at the tip of the torch.



NOTE The Blender file and the world file for the original `porlotex` program were, of course, also created with the Blender world editing system in exactly this same manner. The original files for the `porlotex` world can be found in directory `$L3D/data/levels/tunnel`.

Tips for Working with Attributes

You have now seen all aspects of the Blender portal editing system. Here is a brief list of tips that make working with this system more effective.

- Always check that you are in the proper EditMode/non-EditMode state before clicking any button in `blend_at`. You must be outside of EditMode if you click on Refresh, or in EditMode if you click on New Face Property.
- Always ensure that the current directory and filename are set up properly in Blender and `blend_at`. In particular, if you save your work in another directory or into native Blender format, you must reinitialize the work directory and filename in Blender.
- Only move the magic vertices as a whole; mark all of them with the border select button before attempting to move them. Moving just a few of the magic vertices or edges destroys the overlapping information which encodes the ID, but moving all of them at once preserves the overlap. You would want to move all the magic vertices to reposition the local object coordinate system with respect to the object.
- Never mark any of the magic vertices and click on the Rem Doubles button, since this removes the redundancy on which the entire ID encoding system is built. Be particularly careful when selecting all vertices, since this also selects the magic vertices by default.
- If you suspect that you have indeed accidentally destroyed the magic vertices, then it is best to completely re-tag the mesh. (The symptom of this problem is that `blend_at` finds the wrong attributes file, or no attributes file at all, when Refresh is clicked.) Re-tag the mesh as follows. Select the mesh, enter EditMode by pressing **Tab**, mark all magic vertices, then press **x** and select **Vertices** to delete all the magic vertices. Now, the mesh no longer has any ID associated with it. Exit EditMode by pressing **Tab**. In the `blend_at` window, click the Refresh button. This causes the selected mesh to be re-tagged with a new ID, and to be associated with a new, empty attributes file. At this point, you have lost the original attribute

information, but this information is still stored in the (now unused) attribute file `attribXXX.dat` on disk. You can search for this file manually (perhaps using a file browser, or a tool such as grep to search for a file containing a particular string), and paste its contents into the new attribute file.

- If you duplicate a mesh with **Shift+d**, then you should delete the magic vertices in the duplicate mesh, and re-tag the now anonymous mesh by clicking Refresh in `blend_at`. If you do not do this, then both meshes would have the same ID, which would cause ambiguity when editing the attributes of one of the meshes.
- If Blender appears to act strangely when it is receiving the remote control commands from `blend_at`, one possible reason is that the Alt, Shift, or Ctrl keys have become stuck in Blender, which means that when `blend_at` sends the keystroke a, Blender might incorrectly interpret this as Alt+a. If you suspect that this is the case, then switch to the Blender window, and briefly press Alt twice, Shift twice, and Ctrl twice. This unsticks the keys if they were stuck. Then, try the same operation again in `blend_at`. Notice that, unfortunately, you may have to manually undo some of the incorrect results of the previous remote control session before attempting the operation again.
- If you are working with a display using a color palette, use the PBM utilities to create a common palette for all textures (see Chapter 2).
- `blend_at` does little error checking. If you have saved the directory preferences, they are written to the file `.blend_at.config`. If you later delete the directories referenced by the configuration file, `blend_at` will gracelessly crash. Remove `.blend_at.config` in this case, which causes the directories to be set to reasonable defaults.

Summary of Blender and Portal Worlds

Having now reached the end of our coverage of the Blender portal editing system, let's once again summarize the basic steps necessary to create worlds with this system.

1. Create a project working directory.
2. Start Blender and `blend_at`.
3. If editing an existing world, load the Blender file containing the world.
4. Set up the default load and save filenames for `blend_at` by loading and saving `project.obj` from the project directory.
5. In `blend_at`, ensure that the project directory and working filename are identical to the directory and filename of file `project.obj`.
6. Create sectors as meshes. Use holes in sectors as portals. Portals between sectors must align exactly.
7. Assign each sector a unique ID by selecting the object in Blender, then clicking **Refresh** in `blend_at`. Edit the attributes of each sector in `blend_at` to specify texture information for the faces.
8. Create plug-in objects as cubes.

Team-Fly®

9. Assign each object a unique ID by selecting the object in Blender, then clicking **Refresh** in `blend_at`. Edit the attributes of each object in `blend_at`, to specify plug-in, appearance, and initial sector information.
10. Create the camera as a plug-in object with the attribute `IS_CAMERA YES`.
11. Export the entire world by selecting all objects and exporting to Videoscape. This creates several sequentially numbered files, all using the same base filename specified earlier (`project.obj`).
12. Convert all meshes in the world to the portal world file format with a command of the form

    ```
    perl $L3D/source/utils/portalize/vid2port.pl project.obj* > world.dat
    ```

 and test the resulting `world.dat` file in l3d program `porlotex`. Ensure that all files referenced in `world.dat` are accessible from the executable directory.
13. Save your work to a native Blender file when you are finished with an editing session.

The most important part of these instructions is the default load and save filename. By following the instructions as outlined above, you can use the same filename both for temporary working purposes, as needed by `blend_at`, and for exporting the entire level with Alt+w. This means that if you follow the instructions above, you shouldn't need to worry about the filename while you are editing a world; you can edit, export, and test in a continuous cycle without ever needing to specify a filename.

The only time you do need to worry about the filename is if you save your work in native Blender format (which you should, since Videoscape is best viewed as an export format and not the working format for your data) or in another directory. After such an operation, you must reinitialize the default load and save filename by saving to and loading from the Videoscape project work file `project.obj`. If you do not do so, subsequent save and load operations will operate on the Blender native file—not the Videoscape work file—and the entire `blend_at` program will behave in an undefined manner. If you are ever in doubt, run a quick check in Blender: press **F1, Esc** to glance at the default load filename, and press **Alt+w, Esc** to glance at the default save filename.

Other World Editing Ideas

We mentioned at the beginning of this chapter that the portal-based Blender world editing system is just one of many possible world editing solutions. We purposely looked at this one particular solution in detail, so that you could see all the small problems which need to be solved, and also so that you can experiment with a working system to create levels. Now, let's look at some other possible world editing ideas which could be used for other world organization schemes.

Portalized Regular Spatial Partitioning

As we mentioned in Chapter 5, we can use a regular spatial partitioning scheme combined with a depth-bounded recursive portal traversal. With such a system, our world editor does not need to divide the system up explicitly into sectors. Instead, arbitrary geometry, convex or concave, is just

saved into one or several meshes. It doesn't matter how much or how little geometry is in one mesh. Then, when exporting the world, we automatically slice the world up into equally sized cubes, each of which is a sector. We do this by computing the bounding box for all of the geometry in the world, then dividing this bounding box into smaller boxes. For each of the smaller boxes, we clip all geometry in the world to each plane of the smaller box, to see what part of the world geometry remains inside each box. We saw in Chapter 5 how to clip an object against planes.

After dividing the world up in this way, we then insert portals between all adjacent sectors (cubes). Each interior sector has exactly six portals, one completely encompassing each face of the sector, leading to its neighbors. Not all sectors generated in this way will contain geometry. We can be more efficient by excluding empty sectors from further processing. We could use the `_has_geometry` field of the Perl `Sector` object to keep track of this.

As we saw in Chapter 5, such a portal scheme requires an additional VSD scheme such as a *z* buffer or a painter's sort to produce a correct display at run time. Also, we must always limit the depth of portal traversal with such a scheme, since the portals in this scheme always allow us to see to the end of the world.

The advantage of this approach is that world editing is easier; you don't need to worry about creating separate sectors or aligning portals. The disadvantage is that you have no control over how the geometry gets split into sectors. In the worst case (in fact, in the usual case), a room could get split in half. Then, if you were standing some distance away, the near half of the room would be displayed, but the far half of the room would require a portal traversal beyond the depth limit, meaning that the far half simply would not be displayed until you moved nearer, at which point it suddenly blinks into existence. Therefore, this scheme is less useful for indoor scenes, where it is visually jarring to have parts of rooms suddenly appear or disappear, even if they are far away. This scheme is better suited for outdoor scenes, where appearance or disappearance of hills or trees far away is more tolerable. Combining this scheme with fog to hide the sudden appearance and disappearance of geometry in the distance can also be effective.

BSP Tree and Octree

BSP trees and octrees are similar to the regular spatial partitioning scheme, only hierarchical. Creating a world editor for such a system is also similar. You do not specify any partitioning of space yourself, instead letting the BSP tree or the octree divide the geometry for you. The geometry in the world editor is stored in any convenient manner, divided in any way across an arbitrary number of meshes; then, all geometry is exported at once and is automatically partitioned by the BSP tree or octree. The advantages and disadvantages are similar to the regular spatial partitioning scheme.

Non-convex Sector-based Partitioning

A sort of middle ground between completely manual partitioning and completely automatic partitioning is possible with the use of non-convex sectors. With this scheme, the world editor allows you to divide the world into logical sectors of space, which do not have to be convex. In this case, the partitioning serves to reduce complexity, but not to control explicit visibility information as is the case with convex sectors. For instance, with convex sectors, one logical "room" might have to

be split into multiple physical sectors to preserve convexity. This can be inconvenient from a world editing standpoint, though it makes the visibility computations more efficient. Allowing non-convex sectors permits us to group even non-convex geometry into one logically coherent mesh, but complicates visibility.

To connect the regions, we could use portal polygons, explicitly defining visibility between regions, as we did before. But since our sectors are non-convex anyway, we can also use a more lax portal scheme similar to that with regular spatial partitioning. That is, for each non-convex sector, we compute its bounding box. Then, for each bounding box, we create six large rectangular portals, one on each face of the box and exactly filling that face. For each portal created in this way, we then spatially scan to see which other sectors' bounding boxes overlap the portal. This is an inexact overlap test; if any part of a sector's bounding box overlaps the portal (i.e., intersects with the portal), it counts as overlapping. We connect the portal to the overlapping sectors; there could conceivably be more than one overlapping sector connected to each portal in this case, which requires us to connect all of them to the portal.

Then, at run time, we use the same depth-bounded recursive portal traversal we used for the regular spatial partitioning case. Again, we must depth-bound the traversal because the portals in this case always overestimate visibility; portal connectivity exists between regions even if visibility is actually blocked by intervening polygons.

This sort of world editing scheme requires no scanning for free edges to make portals; instead, it computes bounding boxes, makes large rectangular faces on the portals of these bounding boxes, and connects portals to those sectors which overlap any part of the portal.

Since the sectors are non-convex, it is not easy to determine if a particular point, such as the center of an object, is actually within a sector or not. Also, since portal connectivity is not exact but is overestimated with this scheme, tracking the movement of the camera or an object between sectors is more complicated. We can use the bounding box of the sector as a conservative test to see if a point might be in a sector, but since bounding boxes of sectors can overlap, a point may be classified as being in multiple sectors at once. This requires some scheme to ensure that an object is assigned to just one sector and is processed just once per frame.

The advantage of this scheme is that it allows the world designer control over how geometry gets split into sectors, while simultaneously automating the connectivity among sectors without requiring manual portal specification and alignment. The disadvantage is that portal connectivity is very inexact and overestimated with this scheme, meaning that distant regions of the world still can suddenly blink out of existence if they lie beyond the maximum traversal depth, but at least the world designer can control the exact partitioning of the geometry so as to make areas that "blink out" hidden behind a bending passageway so that the disappearing geometry is hidden behind a wall anyway. Also, this approach requires an additional VSD scheme such as a *z* buffer to produce a correct display, and causes more overdraw than a convex portal scheme (mostly a concern for software rasterizers).

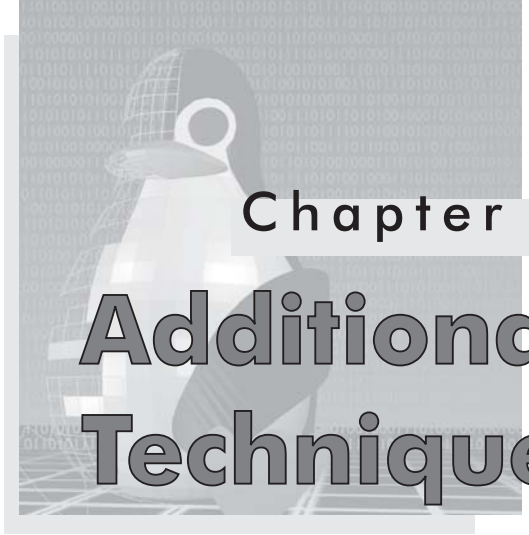


NOTE As far as I understand it, Blender 2.0, which includes game development facilities, uses a scheme similar to this one.

Summary

In this chapter, we looked at a world editing system using Blender. Using Blender and some external tools, we created portal worlds which could be imported into the `porlotex` program from the previous chapter. We saw how to implicitly specify and align portals, how to associate attributes with meshes using binary edge encoding and an external attributes database, and how to use attributes to specify texture and plug-in information. We used the tool `vid2port.pl` to perform the final conversion from the exported Videoscape meshes to the world file. Finally, we looked at some other possible world editing schemes.

Having completed this chapter, we now know how to store, display, manipulate, and create interesting and interactive real-time 3D environments using Linux tools and programs. That's a short sentence, but it summarizes the bulk of this book (and the introductory companion volume *Linux 3D Graphics Programming*). With all of this knowledge, we are now ready for the next chapter, which explores some more advanced 3D graphics techniques.



Chapter 7
Additional Graphics Techniques

The previous chapters have hopefully provided you with a broad and solid understanding of theoretical and practical aspects of Linux 3D graphics programming. In this chapter, we take a whirlwind tour of additional 3D graphics techniques. This chapter is a bit different than the rest of the book in that it does not try to cover any topic in great detail or present lengthy code examples for every concept; instead, this chapter aims to provide you with a perspective on further 3D graphics techniques that you are likely to encounter or need, and that you can explore further on your own. You now know enough to understand all of these topics at a high level; the experience from the previous chapters has equipped you to deal with the low-level implementation issues on your own. You've earned your wings; now it's time to fly.

In this chapter we look at the following topics:

- Techniques for creating special effects, such as environment mapping and lens flare
- Modeling natural phenomena such as smoke, water, or landscapes
- Particle systems
- Level of detail
- Curved surfaces
- Camera tracking in 3D

Special Effects

In previous chapters we have seen all of the basic elements needed to create a reasonably good-looking interactive 3D environment. Let's briefly recapitulate the main techniques we have seen up to this point. Perspective creates a sense of depth and the parallax effect. Arbitrary camera orientation allows for realistic viewpoints. Texture mapping and lighting create engaging surface detail. A 3D modeling program allows us to create believable, complex geometry. VSD schemes combined with a world editing environment enable the creation of entire 3D worlds.

In this section, we now take a look at some other more specialized techniques which can further enhance the visual impact of a 3D program.

Environment Mapping

The term *environment mapping* refers to a special form of dynamic texture mapping used to simulate the reflection of the environment off of a surface. Let's first talk about the general idea, then see how the idea can be applied in real time.

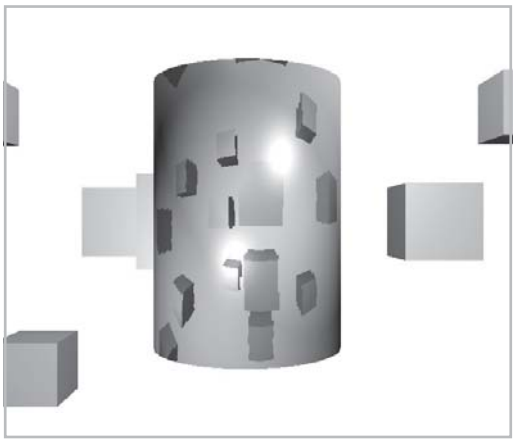


Figure 7-1: A cylinder modeled with environment mapping. The cylinder appears to reflect its environment.

Environment mapping simulates the appearance of a reflective surface by coloring each pixel of the surface's rasterized image with a color coming from a reflected object in the surrounding environment. For a perfect reflection, we could use ray tracing. Understanding the idea behind ray tracing is necessary to understand environment mapping, so let's talk about ray tracing first. With ray tracing, we render reflections as follows.

1. Project an object from 3D into 2D. Rasterize the surface.
2. For each rasterized pixel, determine the 3D point corresponding to the pixel using a reverse projection. Call this point p. From the 3D point, determine the surface normal at that point. Call this normal N. We can interpolate surface normals, as with Phong shading (see Chapter 2), to simulate a smooth curved surface when only flat polygons exist.
3. Using the physical reflection law that the angle of incidence equals the angle of reflection, reflect the vector from the eye to p around the vector N. We can do this using a rotation matrix of 180 degrees, as we saw in Chapter 2. This gives a bounced visibility vector off of the surface.
4. Determine which other object in the environment is the first (nearest) one that intersects the bounced visibility vector. This requires an intersection test between a ray and a polygon, which we cover in Chapter 8. Compute a color and light value for the object at the point of intersection (using texture and light mapping, for instance), and draw this color in the pixel from step 2 currently being rasterized.
5. Repeat for every pixel in the image.

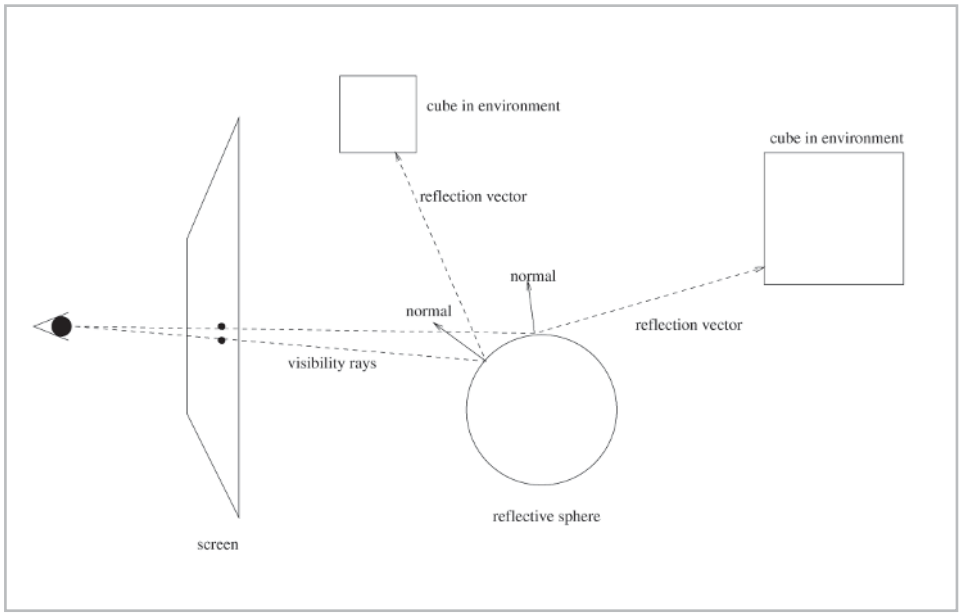


Figure 7-2: Ray tracing to create reflections.

We thus trace rays of visibility from the eye through every pixel in the image, bouncing each ray off of the surface to create a reflection and thereby determining which reflected object is seen for every pixel.

While ray tracing creates good reflections, its problem is that it is slow. Environment mapping is a way of speeding this up by capturing the entire environment beforehand in a special 2D image, called the *environment map*. During the rasterization of each pixel of a reflective surface, instead of searching the entire 3D environment to see which object intersects the reflected visibility vector, we simply look up the appropriate pixel in the 2D environment map. In essence, therefore, environment mapping is a form of texture mapping, but the texture coordinates dynamically depend on the reflected vector from the eye to the surface, correctly causing a change in the reflection whenever the eye moves. Environment mapping assumes that the environment is far away from the reflector, and that the reflector cannot reflect itself (since it is not part of the environment captured in the environment map).

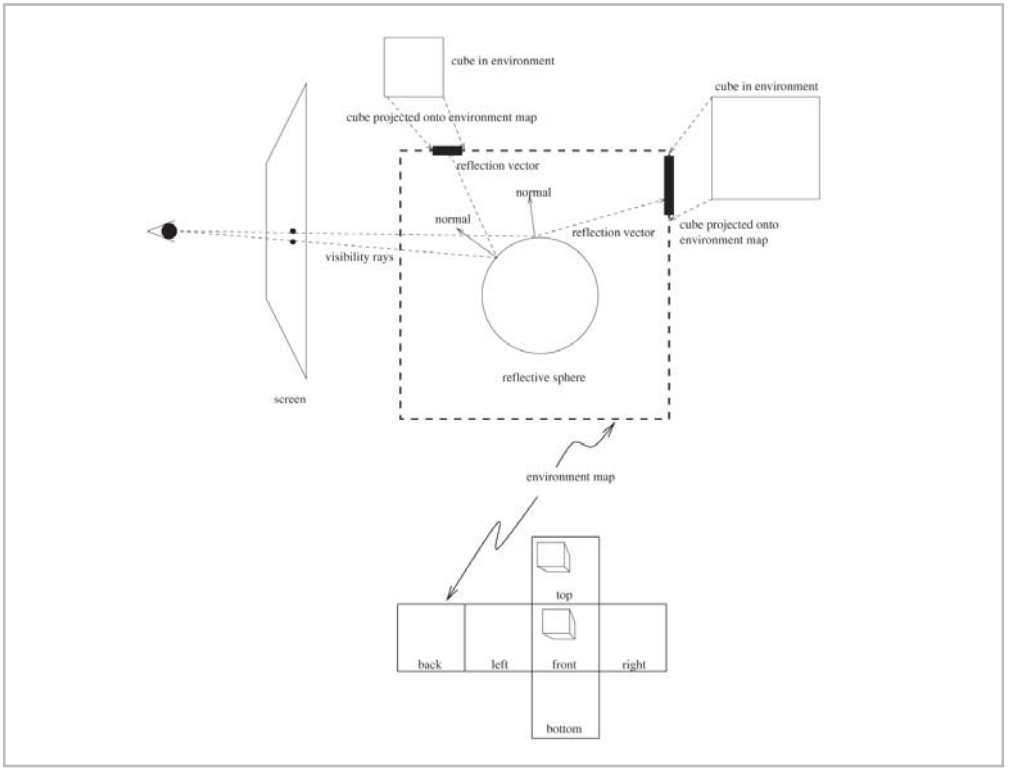


Figure 7-3: Environment mapping first captures the environment in a static 2D environment map, then uses the reflection vector to compute an index into the environment map. Here, cubic environment mapping is shown.

Environment mapping requires us to do two main computations: the computation of the environment map itself, and the mapping of the 3D reflection vector to a 2D (u,v) index within the environment map. The way we compute the environment map affects the way that we map the reflection vector to the environment map. Two common schemes, but not the only ones, are the cubic and spherical environment map.

With the cubic method, illustrated in Figure 7-3, we place a camera in the center of the environment, and construct a cube around the camera. The camera then renders an image of the environment onto each of the six cube faces, each time with a 90 degree field of view. Mapping the reflection vector to the cube is done first by selecting one of the six cube faces based on the largest component of the reflection vector; for instance, a reflection vector of (0.75,0.2,0.63) would select the $-x$ side of the cube, since the x component is largest and is negative. Then, the remaining two components of the normal vector, which both must lie within the range from –1.0 to 1.0 because of the unit length of the reflection vector, are scaled to lie within the range from 0 to 1; for instance, the two remaining coordinates (0.2,–0.63) would map to an environment map (u,v) index (0.6,0.185) on the selected face. By doing this computation for all vertices of a polygon, we can then simply texture map the polygon as normal to make the polygon appear to reflect its

environment. If two vertices of a polygon lie on different faces of the environment map cube, we can split the polygon into two or more pieces, each lying on only one face of the cube.

With the spherical method, the environment map is a circular image representing the appearance of the environment as it would appear orthographically reflected in a sphere. The environment map in this case can be referred to as a *sphere map*. Physically, such sphere maps can be created by taking a photograph of a large chrome sphere with a camera located very far away from the sphere (to simulate orthographic viewing). Algorithmically, we can take a cubical environment map and warp it into a sphere by using image warping techniques, a technique documented in *Advanced OpenGL Game Development* [GOLD98]. Once you have such an environment map, you can then use the following equation to map from the 3D reflection vector r to the 2D (u,v) values in the environment map (the m value is a temporary magnitude value used in the computation and is not needed later):

Equation 7-1

$$\begin{aligned} m &= \sqrt{r_x^2 + r_y^2 + (r_z + 1)^2} \\ u &= \frac{r_x}{2m} + 0.5 \\ v &= \frac{r_y}{2m} + 0.5 \end{aligned}$$

OpenGL directly supports sphere mapping, which means that it can compute these texture coordinates automatically for you by using the above equations. A sample code fragment that performs OpenGL sphere mapping follows.

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_2D);
//- polygon rendering comes here, e.g. glBegin(GL_POLYGON)
glNormal3f(nx,ny,nz);
glVertex3f(x,y,z);
//- etc., glEnd(GL_POLYGON);
```

The calls to `glTexGeni` turn on the OpenGL automatic texture generation, meaning that calls to `glTexCoord*`, which specify explicit texture coordinates, are not necessary; contrast this with the texture mapping Mesa rasterizer of Chapter 2.

For the generation of sphere maps, Mark Kilgard has written a library LibGLsmap, which creates a cubical environment map and then warps it into a spherical map for use by OpenGL sphere mapping. See *Advanced OpenGL Game Development* [GOLD98] for details. Such sphere maps can be generated dynamically, so that whenever the environment changes, the reflections also change. If the environment does not change, a static sphere map could be computed once and used.



NOTE Blender supports rendering images with environment mapping, and was used to create the image of the environment mapped cylinder in Figure 7-1. You must use the TextureButtons in Blender to select an EnvMap texture to apply to a material, which is in turn applied to a mesh. When rendering a scene with environment mapping in Blender, notice that Blender briefly renders six small images, implying that it uses some variation of the cube map scheme. With the SaveEnvMap button in the TextureButtons, Blender allows you to save the computed environment map to a 2D Targa image file (you can convert it to another format with the GIMP program), which contains six small square images, one for each side of the cube. To use this environment map in your program, you would have to implement the cube

mapping yourself, or warp the image into a sphere and use OpenGL's sphere mapping. See the Blender home page `http://www.blender.nl` for tutorials on environment mapping with Blender.

Billboards

A *billboard* is a single texture mapped polygon which always rotates itself so that it directly faces the viewer. Billboards are also often called *sprites*. The texture image on the billboard is typically a 2D image of a more complicated 3D object. For instance, a 2D billboard could contain a pre-rendered image of a complex tree model containing hundreds of polygons. Then, we can approximate the visual appearance of the tree by rendering just one polygon. Thus, the purpose of using billboards is to create the illusion of complex geometry by using a single texture mapped polygon.

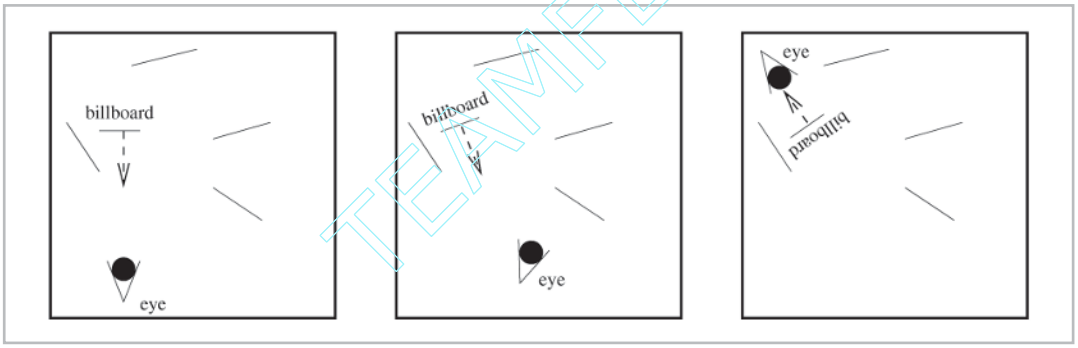


Figure 7-4: A billboard always changes its position so that it faces the viewer.

There are two main ways to define when the billboard "faces the viewer." In one case, the billboard always rotates around all three axes so that it is perfectly parallel to the screen. We call such billboards *screen-aligned billboards*. Such billboards can be used for printing text (which is where the name billboard originally comes from), or for displaying explosions or smoke, as we see below. In the other case, the billboard only is allowed to rotate around one axis, typically its own up axis. We call such billboards *axis-aligned billboards*. This scheme can be used for modeling trees.

To perform the actual rotation for either kind of billboard, we must construct a rotation matrix rotating from the billboard's current orientation to the desired orientation. Think of the billboard as being a camera with its own local coordinate system. The VUP axis points up the billboard, the VRI axis points horizontally along the billboard, and the VFW axis points along the billboard's surface normal. These axes are the billboard's current orientation. Then, express desired orientation in terms of vectors. For the screen-aligned billboard, the desired orientation is the camera orientation, with the same VUP vector but with reversed VRI and VFW vectors. For the axis-aligned billboard, the desired alignment can be expressed as a rotation of the current alignment around the billboard's own VUP vector. Given the desired alignment, we can then construct a matrix converting from the original alignment to the desired alignment.

TEAMFLY
Team-Fly®



Figure 7-5: Screen-aligned billboard. Regardless of the camera orientation, the billboard always faces the camera.

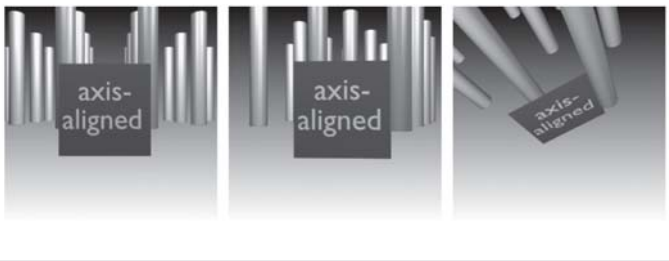


Figure 7-6: Axis-aligned billboard. The billboard is only allowed to rotate about its vertical axis, meaning that a view from above does not cause the billboard to tilt backward.

Typically a billboard will have a so-called *alpha channel*. This is because the billboard polygon is usually rectangular, for simplicity, but the image on the billboard is not generally rectangular. This means that parts of the billboard rectangle are unused. We don't want to draw the parts of the billboard that are unused, because then this would look as if we were sticking rectangular pictures all over the top of our scene—which is in fact what we are doing, but we can make it look better with an alpha channel. An alpha channel is a special type of color that specifies the transparency of an image. We can thus use the alpha channel to set the unused part of the billboard polygon to be transparent and the rest to be opaque.

An alpha channel is typically physically represented in memory by a certain number of bits. For instance, if we have a 24-bit red-green-blue (RGB) color model, with 8 bits each for red, green, and blue, then we can introduce an additional 8 bits for the alpha channel, giving a 32-bit red-green-blue-alpha (RGBA) color model. Graphics acceleration hardware often supports an alpha channel directly, whereas implementing alpha in software manually requires the rasterizer to check for alpha values and combine the new color with the previous color accordingly. (The situation is somewhat easier with just a 1-bit alpha channel, since no color mixing needs to be done; either you see the old color with an alpha of zero, or you see the new color with an alpha of one.)

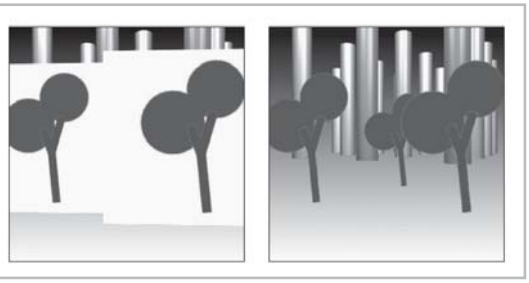


Figure 7-7: An alpha channel allows us to see through the unused parts of a billboard, creating a natural appearance. The billboards on the left use no alpha channel, and it is obvious that they are rectangular textured polygons; the billboards on the right use an alpha channel.

Billboards can be used in combination with other techniques to create visual special effects, with fewer polygons than would otherwise be required. We'll now look at some of these techniques.

Lens Flare

Lens flare is a phenomenon caused by the imperfection of real-world lenses when confronted with very bright light, causing a sparkle or flare to appear in the final image. We can use billboards to simulate lens flare, thereby giving or underscoring the appearance of bright light in an image. We simply render or draw a 2D image texture that looks like a small sparkle, then draw this texture onto a screen-aligned billboard. We can render the lens flare after all other geometry is rendered, so that the lens flares never appear to be obstructed by other geometry. Such an obstruction would make it clear that the lens flare is merely a textured polygon.

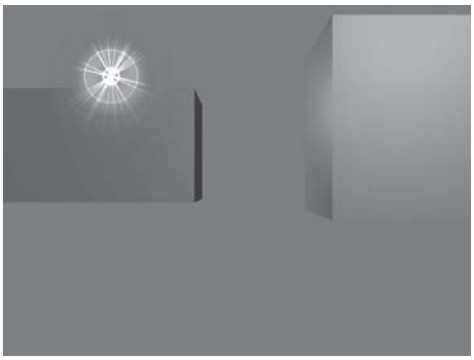


Figure 7-8: Using a billboard to simulate lens flare.

We can control the drawing of the lens flare with line-of-sight and distance parameters. For instance, we can model a bright light source as being behind a wall. If a straight line of sight from the camera to the light source exists, we can draw the lens flare billboard; otherwise, we do not draw it. This creates the desired effect of having the lens flare suddenly pop into existence as soon as the light is visible. We can also draw the lens flare billboard larger as we move closer to the light source, and smaller as we move away, by using a distance formula. Although the lens flare would already automatically increase in apparent size as we approach it due to normal perspective, we can exaggerate this effect by making the lens flare grow or shrink in size exponentially as we approach or recede from it. We would want to exaggerate the growth or shrinkage of the lens flare with distance because the lens flare should not appear to be a physical point in space that we can normally approach; it is instead a lens artifact that depends on our viewing position.

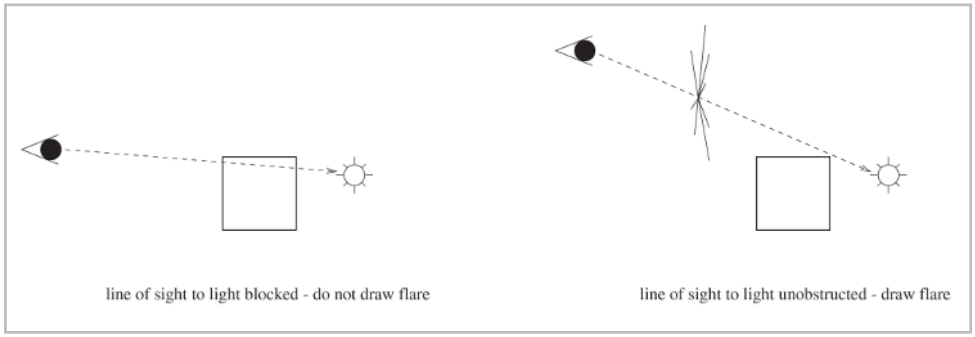


Figure 7-9: Using line-of-sight to control lens flare.

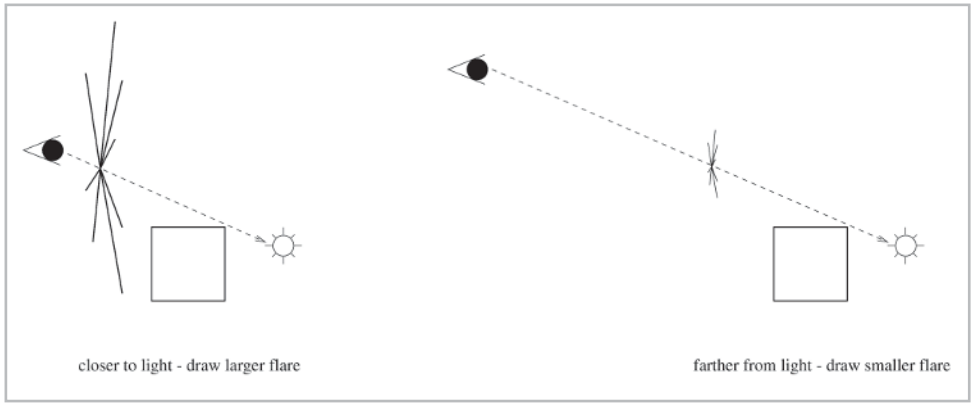


Figure 7-10: Using distance to control lens flare.

Particle Systems

A *particle system* is a system of many small objects, called *particles,* all of which are set into motion and controlled by some algorithm specific to the system. Any effect which can be viewed as a collection of particles can be modeled with a particle system. Examples of systems that can be modeled with particle systems are explosions, fire, smoke, a swarm of insects, or a school of swimming fish.

The particles in a particle system are typically numerous, ranging from the hundreds to the thousands. It is the collective optical effect of all the particles as a whole system that leads to the term particle system. Because the effect depends on the quantity of particles, it is often the case that the particles themselves are very simple objects. They can be actual polygonal objects (as with a school of fish), or simple billboards (for fire or smoke), or even simply single points (for explosions or insect swarms). The simpler the representation, the more particles you can have in the system. Speed is also of primary concern in a real-time particle system, because there are so many particles.

Particle systems are particularly fun to program and play with because with just a few lines of code, you can create an interesting and, as we now see for the first time in this book, physically realistic system. Let's now look at a simple particle system.

Physics and Particle Systems

The idea behind our particle system is amazingly simple, yet it produces quite satisfying results. The next sample program `particle` simulates an explosion of a few thousand particles. We start all particles out at a certain position, and then impart each particle with a random 3D velocity away from the starting point, to simulate the tremendous and practically instantaneous outward force generated by an explosion. From classical Newtonian physics, we know that velocity is a change in position over time. The velocity of each particle, therefore, is expressed as a 3D vector indicating the *x*, *y*, and *z* displacements per second for each particle. We'll see in a moment exactly how the time component, the seconds, plays a role.

This alone would be enough to generate an explosion effect, but it would be mostly uninteresting—at least, in comparison to all the things we can achieve with a particle system. In particular, the velocity of each particle would not change over time with this system. This may be the way things work in outer space, where no other forces act on the particles, but let's make it a bit more interesting and assume we are modeling an explosion of particles on the earth.

On Earth, all bodies are subject to the force of gravity. Objects fall toward the ground with an acceleration of 9.8 m/s^2 (meters per second squared), regardless of their mass. Acceleration is a change in velocity over time; velocity, in turn, changes position over time. This means that if we are modeling an explosion on Earth, each particle's velocity should, each second, increase by 9.8 m/s downward toward the earth. This creates a realistic falling effect. The acceleration is expressed as a 3D vector indicating the *x*, *y*, and *z* change to the velocity per second. Assuming our left-handed coordinate system with *y* going up, this means that our acceleration vector due to gravitational force is (0,–9.8,0) meters per second, each second.

To make the simulation more interesting, we also model a flat, hard floor off of which the particles can bounce. If a particle is found to have moved beneath the floor, we correct its position to be back on the surface of the floor, and reverse the *y* component's velocity so that it is no longer traveling down, but up. We also reduce the *y* component of the velocity to simulate a loss of energy after impact with the floor. Note that this handling of collision with the floor is quite simplified; see the section on collision detection in Chapter 8 for some of the issues involved with more accurate collision detection and response.

Finally, we give each particle a lifetime in seconds. After a certain number of seconds, the particle "dies" and is no longer visible. In this sample program, when a particle dies, we reinitialize it so that the particle system is always in motion. Furthermore, all particles are initially created simultaneously and have identical lifetimes, meaning that the explosion animation appears to repeat itself because all particles die and reincarnate with one another simultaneously. You could give each particle a random lifetime to create a more continuous, non-repetitive flow of particles, but then it wouldn't look so much like a massive explosion, rather more like a stream of individual, high-powered particles being shot one after another. The choice of lifetime depends on the desired effect.

Real-Time Update

We've now laid down the simple physical rules governing our particle system, which are a simplification of Earth's rules. Our particles start out with a random initial explosive outward velocity, which changes the particles' positions over time. All particles are subject to the force of gravity, which causes an acceleration of the particles in the downward direction. This acceleration changes the particles' velocities over time, again changing the position over time. Particles die and are reborn after a fixed lifetime in seconds.

The one new component here is the time factor. All of our dynamic quantities here depend on elapsed time in seconds. Until now, we have not used time explicitly in the update routines for our objects. We updated objects from frame to frame, with the knowledge that one frame temporally follows the previous one, but we didn't know exactly how much time elapsed since the last frame. Until now, this was largely unimportant, because we weren't doing any physically based simulation. Now we are. In Chapter 1, we made a passing comment on this phenomenon, repeated here:

"The calling frequency of `update_event` is not necessarily guaranteed to be constant. That is to say, the amount of physical time which elapses between successive calls may be slightly different. For accurate physical simulations, where velocities or other physical quantities should be updated based on time, we can store an internal variable recording the value of the system clock the last time that `update_event` was called. We can then compare the current system clock to the value of the variable to determine how much physical time has elapsed, and update the time-dependent quantities accordingly."

This explains exactly what we are going to do in our sample program to update our positions and velocities with respect to time. We update each particle individually. For each particle, at the beginning of a frame, we use the system call `gettimeofday` to retrieve the current system time. We then compare the current time to a previously saved time value, which was set during the previous frame at exactly the same position in the code. We subtract the current time from the previous time see how much physical time, in seconds, has elapsed since the last frame. Typically this value will be a fraction of a second (and should be, for interactive 3D applications).

Given the elapsed time since the last frame, we multiply it with the time-dependent quantities to obtain the change in the time-dependent value, and update the time-dependent values accordingly. For instance, let's say we have a particle with an initial velocity of 10m/s in the positive *y* direction. This means that the particle is traveling upwards. Let's say that 0.25 seconds elapsed since the last time we called this routine. First, we determine the change in distance, by multiplying the elapsed time with the velocity: 10m/s*0.25s = 2.5m. This means that in the 0.25 elapsed seconds since the last time we updated the particle's position, the particle has traveled 2.5 meters. Then we update the velocity itself. Due to gravity, acceleration changes the *y* component of velocity by –9.8m/s each second. We multiply the acceleration by the elapsed time: 0.25s* ($-9.8m/s^2$), yielding a change in velocity over those 0.25s of –2.45m/s. We add this change in velocity to the current velocity, giving us 10m/s–2.45m/s = 7.55m/s as the new current velocity of the particle. Now, the particle is still traveling upwards, but more slowly, because gravity is pulling it down. After performing these computations, we save the current time into a variable, which represents the last time that we took physical action on our particle. The next time the routine is called, the new position and velocity are again updated based on the elapsed time.



NOTE This means of computing new values of time-dependent quantities through multiplication with the elapsed time is called *Euler integration*. It is not completely correct, because it assumes the particles were moving in straight lines with constant velocity and acceleration during the elapsed time. This is generally not the case, which leads to cumulative errors in the computed positions, velocities, and accelerations of the particles. See the section on physics for more information.

Sample Program: particle

With the preceding explanation, understanding the source code listing for the sample program `particle` should be easy. The structure of this program is based on that of the `dots` program presented in the introductory companion book *Linux 3D Graphics Programming* (and included on the CD-ROM). We have mercilessly slashed out the code for the runway lights, stars, and snow examples, and focused the program on the particle system simulation. The abstract class `mydot` has been extended to include a member variable `lifetime`, indicating how much longer this dot has to live, in seconds, and a method `is_alive`, returning if the dot is still alive or not.

The pipeline class has been minimally changed, so that instead of determining if a dot is outside of the world coordinate bounds, as we did in the original `dots` program, we check to see if the dot is alive or not. If not, then we reset it.

The new `dot` subclass `dot_particle` controls the behavior of the particles, which in this program are simply drawn as dots. The methods of interest are `place_dot` and `move_dot`. Method `place_dot` places a dot at the origin, resets its lifetime, and assigns it a random outward velocity. This method is called to initially place the dots at program start and to resurrect a dot whenever it dies. Method `move_dot` computes the elapsed time since the last invocation of `move_dot`, and updates the position and velocity accordingly. If a dot is found to be beneath the floor, it is repositioned to lie on the floor, and its velocity in the y direction is instantaneously negated and scaled down slightly, simulating a vertical bounce with loss of energy; gravity eventually pulls the dot back down again. Also, we similarly bounce the dots off of invisible, vertical, axis-aligned planes, if the dot's x or z values are too large or small. Finally, we reduce the lifetime of the dot so that it eventually dies and gets reborn.

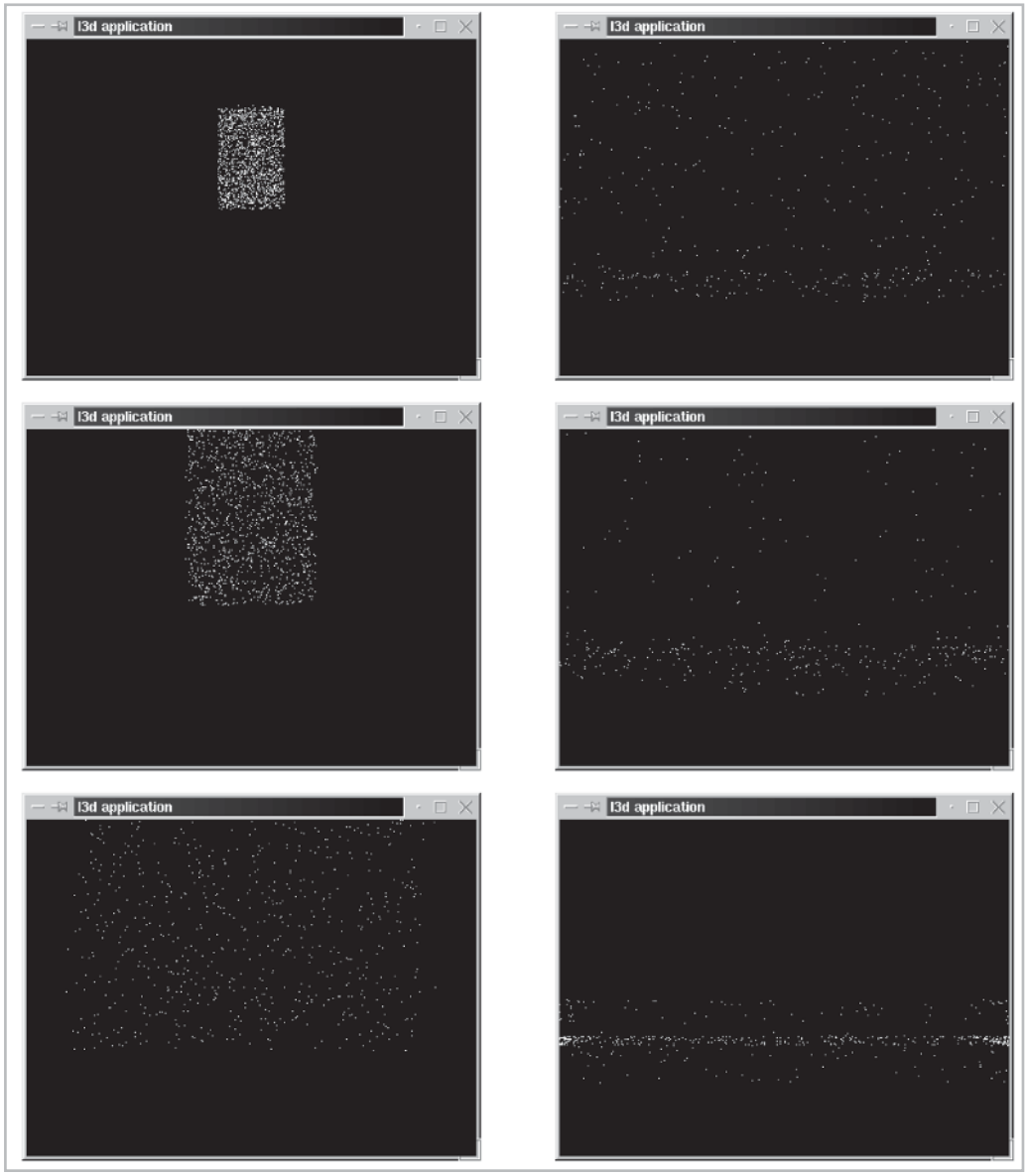


Figure 7-11: Output from sample program `particle`.

Listing 7-1: `particle.cc`

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
```

```
#include "../lib/raster/rasteriz.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/system/fact0_3.h"

#include "../lib/geom/point/point.h"
#include "../lib/math/vector.h"
#include "../lib/math/matrix.h"

#define MAXDOTS 1500

#define PI 3.14159265

void choose_factories(void) {
  factory_manager_v_0_3.choose_factories();
}

l3d_real fovx, fovy;
int xsize, ysize;

class mydot {
  public:
    l3d_coordinate position;
    l3d_vector displacement;
    l3d_vector velocity;
    mydot(void) {position.original.set(int_to_l3d_real(0),
                                       int_to_l3d_real(0),
                                       int_to_l3d_real(0),
                                       int_to_l3d_real(1)); }
    virtual void place_dot(void) = 0;
    virtual void move_dot(void) = 0;
    virtual void project_dot(void) {
      position.transformed.X_ =
        l3d_mulri(l3d_mulrr(l3d_divrr( position.transformed.X_,
                                       position.transformed.Z_),
                            fovx),
                  xsize)
        +
        int_to_l3d_real(xsize > 1);

      position.transformed.Y_ =
        l3d_mulri(l3d_mulrr(l3d_divrr(-position.transformed.Y_,
                                      position.transformed.Z_),
                            fovy),
                  ysize)
        +
        int_to_l3d_real(ysize > 1);

    }
    virtual int is_alive(void) = 0;
};

class my_pipeline : public l3d_pipeline {
  protected:
    l3d_rasterizer_2d_imp *ri;
    l3d_rasterizer_2d *r;

    unsigned long color;

    mydot *dots[MAXDOTS];
```

```
    int numdots;

    long i, j;
    long wc_win_width, wc_win_height, wc_win_dist, fov;
    long wc_win_left, wc_win_top;

  public:
    l3d_screen *s;
    my_pipeline(void);
    virtual ~my_pipeline(void);

    void key_event(int ch);
    void update_event(void);
    void draw_event(void);
};

my_pipeline::my_pipeline(void) {
  s = factory_manager_v_0_3.screen_factory->create(400,300);
  ri = factory_manager_v_0_3.ras_2d_imp_factory->create(400,300,s->sinfo);
  r = new l3d_rasterizer_2d(ri);

  s->sinfo->ext_to_native(0,0,0);
  s->sinfo->ext_max_red =
    s->sinfo->ext_max_green =
      s->sinfo->ext_max_blue =  255;
  color = s->sinfo->ext_to_native(255, 255, 255);
  s->refresh_palette();

  xsize = s->xsize;
  ysize = s->ysize;
  fovx = float_to_l3d_real(1.0/tan(50./180. * PI));
  fovy = float_to_l3d_real(1.0/tan(30./180. * PI));

}

my_pipeline::~my_pipeline(void) {

  int dot_i = 0;
  for(dot_i=0; dot_i<numdots; dot_i++) {
    delete dots[dot_i];
  }

  delete s;
  delete ri;
  delete r;
}

void my_pipeline::key_event(int ch) {
  switch(ch) {
    case 'q': {
        exit(0);
      }
  }
}

void my_pipeline::update_event() {
  int i;

  for(i=0;i<numdots;i++) {
    dots[i]->position.reset();
```

```
    dots[i]->move_dot();
    if(! (dots[i] -> is_alive())) {
      dots[i]->place_dot();
    }
    dots[i]->project_dot();
  }
}

void my_pipeline::draw_event(void) {
  int i;
  r->clear_buffer();

  for(i=0;i<numdots;i++) {
    if(dots[i]->position.transformed.X_>=0 &&
        dots[i]->position.transformed.X_ <= int_to_l3d_real(s->xsize) &&
        dots[i]->position.transformed.Y_ >= 0 &&
        dots[i]->position.transformed.Y_ <= int_to_l3d_real(s->ysize))
    {
      r->draw_point(ifloor(dots[i]->position.transformed.X_),
                    ifloor(dots[i]->position.transformed.Y_),
                    color);
    }
  }

  s->blit_screen();
}

class dot_particle : public mydot {
  protected:
    l3d_vector accel;
    float lifetime;
    long last_sec, last_usec;
  public:
    void place_dot(void); //- virtual
    void move_dot(void); //- virtual
    int is_alive(void);
};

void dot_particle::place_dot(void) {
  displacement.set(0,
                   0,
                   50,
                   0);
  position.reset();
  accel.set (float_to_l3d_real(0.0),
             float_to_l3d_real(-9.8),
             float_to_l3d_real(0.0),
             float_to_l3d_real(0.0));

  velocity.set(float_to_l3d_real( l3d_divri( int_to_l3d_real(rand()%500),10) - int_to_l3d_real(25) ),
               float_to_l3d_real( l3d_divri( int_to_l3d_real(rand()%500),10) ),
               float_to_l3d_real( l3d_divri( int_to_l3d_real(-rand()%500),10)),
               float_to_l3d_real(0.0));
  lifetime = 15.0;

  struct timeval tv;
  struct timezone tz;
  tz.tz_minuteswest = -60;
  gettimeofday(&tv,&tz);
  last_sec = tv.tv_sec;
```

TEAM-FLY

```
  last_usec = tv.tv_usec;

}

void dot_particle::move_dot(void) {

  struct timeval tv;
  struct timezone tz;
  tz.tz_minuteswest = -60;
  gettimeofday(&tv,&tz);

  l3d_real delta_t = float_to_l3d_real
                     ( (tv.tv_sec + 0.000001 * tv.tv_usec -
                        (last_sec + 0.000001 * last_usec)) );

  displacement = displacement + velocity*delta_t;

  l3d_vector total_accel;
  total_accel = accel;
  velocity = velocity + total_accel*delta_t;

  if(displacement.a[0] > int_to_l3d_real(100)) {
    velocity.a[0] = l3d_divri(-velocity.a[0],4);
    displacement.a[0] = int_to_l3d_real(100);
  }
  if(displacement.a[0] < int_to_l3d_real(-100)) {
    velocity.a[0] = l3d_divri(-velocity.a[0],4);
    displacement.a[0] = int_to_l3d_real(-100);
  }
  if(displacement.a[2] > int_to_l3d_real(120)) {
    velocity.a[2] = l3d_divri(-velocity.a[2],4);
    displacement.a[2] = int_to_l3d_real(120);
  }
  if(displacement.a[2] < int_to_l3d_real(60)) {
    velocity.a[2] = l3d_divri(-velocity.a[2],4);
    displacement.a[2] = int_to_l3d_real(60);
  }
  if(displacement.a[1] < int_to_l3d_real(-10)) {
    displacement.a[1] = int_to_l3d_real(-10);
    velocity.a[1] = l3d_divri(-velocity.a[1],4);

  }
  position.reset();
  position.transformed = position.transformed + displacement;

  lifetime -= delta_t;

  last_sec = tv.tv_sec;
  last_usec = tv.tv_usec;

}

int dot_particle::is_alive(void) {
  return (lifetime > 0.0);
}

class my_pipeline_particle : public my_pipeline {
  public:
    my_pipeline_particle(void) {
```

```
      int i;

      for(i=0; i<MAXDOTS; i++) {
        dots[i] = new dot_particle();
        dots[i]->place_dot();
      }
      numdots = MAXDOTS;
    }
};

int main(int argc, char **argv) {

  choose_factories();

  l3d_dispatcher *d;
  my_pipeline *p;

  d = factory_manager_v_0_3.dispatcher_factory->create();
  p  = new my_pipeline_particle();

  d->pipeline = p;
  d->event_source = p->s;
  d->start();

  delete d;
  delete p;
}
```

Comments on the Sample Program's Physics

There are actually several oversimplifications made to the physics modeling in this program. Just for starters, the particles have no mass or volume, and cannot interact with each other. Also, we wantonly fooled around with the particles' positions and velocities whenever we found that they happened to be beneath the floor, instead of actually detecting and handling collisions. So, although each particle behaves in a mostly realistic manner, there are still several physics issues which have been vastly and incorrectly simplified in this example. (We briefly return to the topic of physics in Chapter 8.) This is one of the advantages of particle systems: you can use a simple particle system as a testbed for simple physics code, and even if the physics isn't completely correct, with a particle system it still tends to look good due to the sheer number of particles. Handling physics correctly is more important when we have a few, large objects interacting with one another, when we are more likely to notice exactly how the objects behave.

Some Ideas for You to Try

You might want to experiment with the particle system to try to create some more interesting effects. Some ideas are:

- Allow the particles to interact with one another. For instance, assign each particle a magnetic attractive or repulsive force, which affects all other particles. This would require saving the previous state of all particles in an array, and then updating each particle based on the sum of all previous forces exerted by all particles.

- Assign each particle a color based on its lifetime. This could be used for a fire effect, where newer particles glow with a hot white color, while older particles fade away to dull red.
- Add extra forces such as air resistance, negative gravity, and additional gravitational centers.
- Model spring forces between the particles to build more complex structures.
- Create real `l3d_object` objects or billboards instead of dots to represent the particles.
- Allow for arbitrary camera movement so you can observe the particle system from any angle.
- Encapsulate a generalized particle system into a class, and write a plug-in object that uses the particle system. Examples of such plug-in objects could include flamethrowers, explosions, or water fountains.

Natural Phenomena

There are a number of techniques we can use to simulate the appearance of natural phenomena, such as water, clouds, smoke, and so forth. Since real physical modeling and simulation of these phenomena is currently enormously computationally expensive, we have to find ways of simulating their appearance. Simulating natural phenomena mostly relies on a clever use of texture mapping, billboarding, and sometimes use of actual polygonal geometry.



NOTE Real numerical simulation of such natural phenomena is, of course, possible. For instance, I used to work at a meteorology lab, where thunderstorms were numerically simulated in 3D using computational fluid dynamics techniques, sometimes in real time. It was a routine matter to run these simulations on a Cray or a cluster of RISC workstations. This shows the sort of computing power which is needed to run such simulations in real time—and "real time" here merely means faster than the weather, not at interactive frame rates with multiple updates per second. So unless you have a spare Cray lying around, you'll have to be satisfied with simulating the appearance rather than the real behavior of natural phenomena.

Let's now look at some kinds of natural phenomena whose appearance we can easily simulate. *Advanced OpenGL Game Development* provides more coverage of some of these techniques [GOLD98].

- Clouds: We can create a single, large polygon with a repeating, small cloud texture, and place it high within the scene. By dynamically changing the texture coordinates, the cloud images will appear to drift across the cloud layer. We can use multiple cloud polygons at different heights in combination with an alpha channel to create the appearance of multiple cloud layers. The cloud texture can be created manually or by some procedural or fractal method. Image manipulation programs such as the GIMP often have plug-ins for creating cloud-like and other natural textures; Blender also has cloud support as a procedural texture.

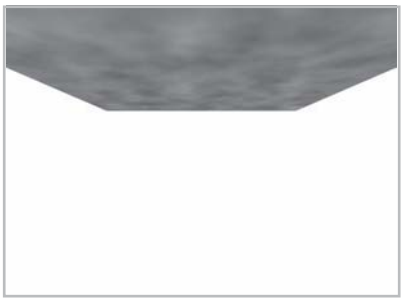


Figure 7-12: A polygonal cloud layer.

- Sky domes and sky boxes: Instead of using a single polygon for the sky, we can surround the camera with a huge texture mapped dome or box. The box is large enough so that its faces all appear very distant from the camera. We then map any static background geometry onto the faces of the box—for instance, distant mountains, clouds, and a fiery sunset. The box is drawn in camera space with unchanging coordinates. This causes all parts of the box always to be a certain fixed distance from the camera; the viewer can never get physically closer to the background image polygons. This produces the illusion of an infinitely far background.
- Smoke: A single screen-aligned billboard polygon with an alpha channel can be used to simulate a single puff of smoke. Over time, we can make the billboard larger and more transparent to simulate the expansion and dissipation of the smoke. By using several smoke puffs one after another, each independently expanding and eventually fading completely away, we can create a trail of smoke. Smoke textures, like cloud textures, can be generated algorithmically or by hand.

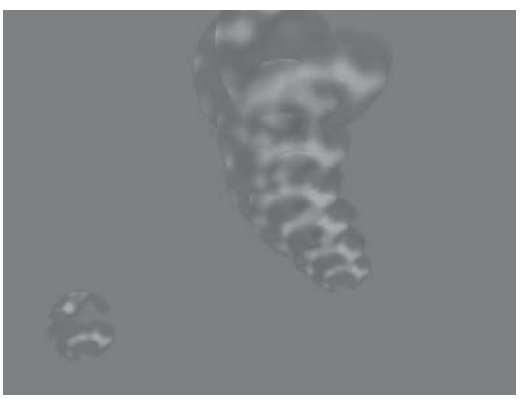


Figure 7-13: Smoke puffs. The billboard image used for the puff can be seen in the lower left of the figure.

- Fire: An animated series of screen-aligned billboard polygons can create the illusion of a flickering fire. The same technique can be used for explosions and fireballs. We can combine these techniques with the smoke techniques.



Figure 7-14: Fire. These images were created automatically with the GIMP image manipulation program (using the menu options Filters-Render-Flame).

- Water: We can use a single translucent, colored polygon with a changing ripple texture to simulate water, such as the surface of a pool. The geometry underneath the water then appears to be seen through the polygon representing the water surface. We can treat this translucent water surface polygon as a portal, possibly distorting the underwater geometry behind the portal in a slow, cyclic fashion, to simulate the changing refraction of light through moving water. We can also use a textured polygon mesh, which is dynamically morphed over time with sine waves to geometrically model the surface of the water and waves. Environment mapping can be used to capture reflections in the water; if the water is perfectly still, a simple second rendering of the reflected geometry (as with portal mirrors; see Chapter 5) can be used for the reflection. Underneath water, we can use some of the following techniques: a pale underwater green color, upward-floating air bubbles (either billboards or polygonal, translucent spheres), and wavy image distortion of the 2D image just before rasterization.

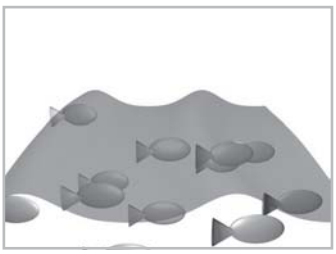


Figure 7-15: Water. Here, we use a textured, wavy mesh with an alpha channel to simulate the water surface. For clarity, no surrounding geometry is drawn around the water mesh, so you can see directly beneath the water polygons to the fish. Ordinarily, the water mesh would be completely surrounded by land geometry, so that the only way to see the underwater objects would be through the translucent water mesh.

- Fog: We can also use the fog techniques of Chapter 2 in combination with any or all of the above.

Bump Mapping

Bump mapping is a technique used to give a flat surface the appearance of having small surface perturbations, or bumps and pits. Bump mapping is similar to texture mapping in that it uses a 2D image map, the *bump map*, whose contents are used to vary the surface normal of a polygon during rasterization on a per-pixel basis. The modified surface normal is then used to compute lighting. Because the lighting computation depends on the surface normal, the modified surface normal gives the appearance that the surface has been perturbed slightly, leading to small shadows or highlights.



Figure 7-16: Bump mapping.

One way of using the bump map is to store a (u,v) displacement at each location in the bump map. We then add these vectors to the tip of a real surface normal vector to push or pull the tip of the normal into a new direction, much as you would push or pull the tip of a joystick to change its direction. Another way to use the bump map is to interpret the bump map as a height field, where each pixel in the bump map is a height value. In this case, the bump map then completely determines the surface normal, instead of modifying the existing surface normal. The slope between any two adjacent bump map pixels determines the surface normal; if two adjacent pixels in the bump map have the same height, then the slope between them is flat, and the normal points straight up; if they have a different height, then the slope points either upward or downward, and the normal is perpendicular to the slope.

Bump mapping as described here is a computationally expensive operation, since it requires normal and lighting calculations for each pixel, but there are a number of ways of speeding it up. *Real-Time Rendering* by Tomas Möller and Eric Haines provides an overview of several methods [MOEL99].

Multi-pass Techniques

The recent increase in availability of graphics acceleration hardware has made it feasible to implement *multi-pass* rendering algorithms. Multi-pass algorithms render the same scene multiple times, combining the results of each rendering into the final 2D image which gets displayed on-screen. This, of course, implies increased rendering work. There are two main reasons for multi-pass rendering: either it is the most natural solution to a problem, or the presence of hardware acceleration makes it the fastest solution to a problem.

We've actually already seen an example of multi-pass rendering in Chapter 2, where we used multi-pass rendering with Mesa to modulate our textured polygons with the light coming from light maps. This is an example of where the presence of hardware acceleration makes multi-pass rendering the fastest solution to a problem. In software light map rendering, recall, we computed a surface, which was the combined light map and texture map. Then, we drew this combined surface in one pass, using an unmodified texture-mapping rasterizer. Computing the surface costs time and memory. With hardware acceleration, it is more efficient just to make two passes on the scene, then combine them. Combining the two passes can be done with the function `glBlendFunc`, as we saw in Chapter 2.

An example of multi-pass rendering as the most natural solution to a problem is when multiple, dynamic lighting components interact. Consider an example of a textured and environment

mapped model of a shiny apple. The texture map represents the surface of the apple itself; it would be red and slightly uneven to simulate the appearance of the apple peel. The environment map represents the reflection of the environment off of the apple's surface. If we move the viewpoint or the apple, then the apple's texture (the peel) appears to remain fixed to the apple's geometry, but the reflections, since they depend on the viewpoint, appear to change with respect to the apple's geometry. Thus, we have two different images which get mapped onto the apple's geometry, each of which is computed differently than the other. The texture map's coordinates are fixed; the environment map's texture coordinates change with the viewpoint, but both ultimately affect the same 2D pixels resulting from the projection of the same 3D points. While it would theoretically be possible to combine the environment map and the texture map into a single texture, as we did with light mapping, this would need to be redone for every change in the viewpoint, since the environment map contribution changes based on the viewpoint, unlike the diffuse lighting components (which are viewpoint independent) captured into a static light map. In this case, therefore, it is most natural to render the texture in one pass, the environment in another pass, and blend the two images together.

OpenGL 1.3 supports *multi-texturing*, where the graphics API itself (and possibly also the hardware) allows for the simultaneous specification of multiple texture coordinates for one set of geometry. Then, in the presence of appropriate multi-texturing hardware, the scene is rendered with multiple textures, but in the time it takes to render just one pass.

Advanced Techniques

In this section, we take a look at some graphical techniques which use interesting algorithms to create complicated or visually interesting graphics. The advanced techniques of this section differ from the special effects of the previous section in that the special effect techniques generally relied upon a fairly simple idea, with the goal of generating a visually interesting result. On the other hand, the advanced techniques in this section generally have somewhat more complex algorithms, and often have a goal other than just a visual glamour. They can represent an entirely different or new way of handling some fundamental aspect of 3D graphics.

Curved Surfaces

The focus of this book has been on polygonal 3D graphics. We stored our models as polygons, and rendered these polygons to the screen.

An alternative is to represent the geometry in terms of curved surfaces, then to tessellate the geometry dynamically to generate a polygonal representation. While it would theoretically be possible to render the curved surface directly by using a z buffer (remember, the z buffer only requires us to be able to compute a z value for the surface for every pixel), it is generally more common to tessellate the curve into polygons, so that hardware acceleration for polygons can be exploited.

There are a number of curved surface representations. Hermite surfaces, Bezier surfaces, and NURBS (Nonuniform Rational B-Spline) surfaces are some of the more common ones. A *patch* is

a small segment of a curved surface; think of a patch as a square sheet of rubber which has been stretched into a curved shape. These curved surface representations are examples of parametric, bicubic surfaces. This means that we obtain an (x,y,z) point on the surface by evaluating a parametric function $f(u,v)$, where u and v are parameters going from 0 to 1 representing the relative position along each side of the patch. The function f which returns the (x,y,z) is cubic in u and v, meaning that it is a polynomial including terms up to u^3 and v^3.

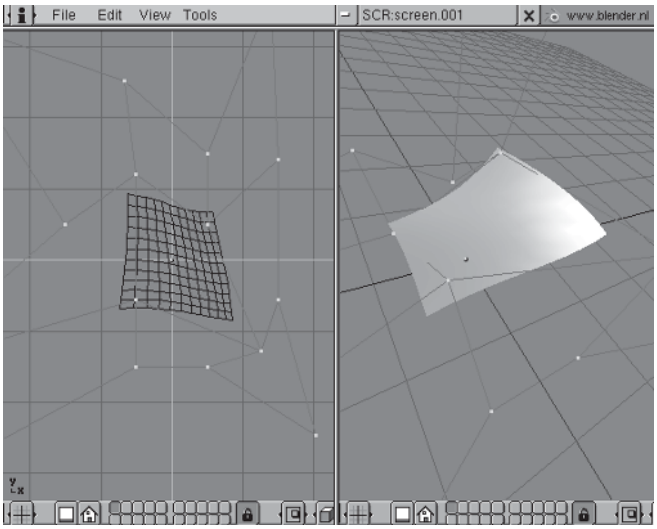


Figure 7-17: A single NURBS patch modeled in Blender.

To render a patch as polygons, we can evaluate successive rows on the patch. For instance, we would keep v constant, say 0.0, then evaluate $f(u,v)$ for u from 0.0 to 1.0 in any chosen interval, say ten steps. Then we increase v slightly, for instance to 0.1, and evaluate $f(u,v)$ again for u from 0.0 to 1.0. This gives us two rows of vertices. By connecting the vertices in a zig-zag fashion, we create a series of triangles, which can then be rendered as polygons. A series of triangles connected in this way is called a *triangle strip*. Some graphics acceleration hardware can draw triangle strips faster than drawing individual polygons, because in the interior of the strip, a single new vertex defines a new triangle; the other two vertices are reused from the previous triangle in the strip.

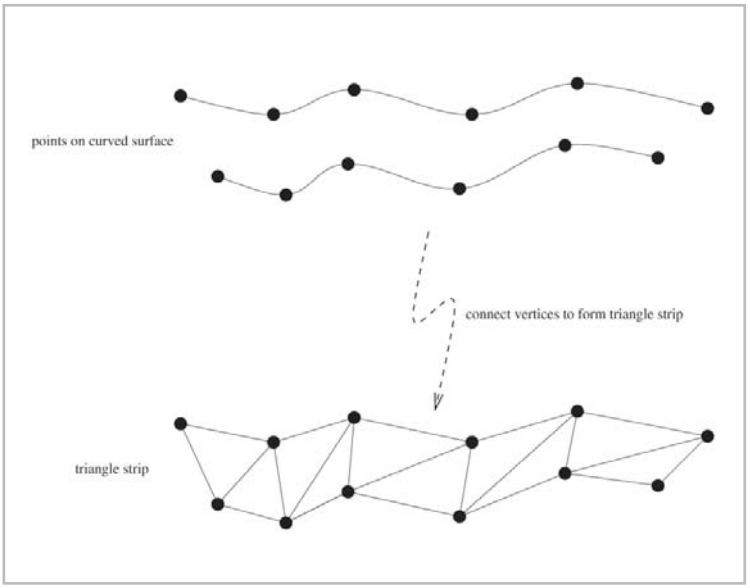


Figure 7-18: Using adjoining rows of vertices to create a triangle strip.

Importantly, notice that we can choose to create anywhere from a few to infinitely many polygons to approximate the curved surface. We simply adjust how finely we step along the parameter axes *u* and *v*. In this way, curved surfaces present a scalable solution. If we have processing power to burn and appropriate hardware acceleration, we can step along *u* and *v* in increments of 0.00000001 to create beautifully smooth surfaces with millions of polygons. If we're running on a less well-equipped system, we step more coarsely and produce fewer polygons. Using curved surfaces allows us to store just one representation of the geometry in memory, namely, the parametric bicubic function *f*, but still gives us the flexibility to create multiple polygonal representations of that geometry at varying levels of detail. The next section covers other level of detail techniques.

Another interesting curved surface representation is that of *subdivision surfaces*. Subdivision surfaces begin with a normal polygonal mesh, then subdivide the faces of the mesh to add new faces and vertices. The new vertices are then moved to new positions based on some guiding rules; in particular, the new vertices are positioned such that the resulting faces are curved, where curved is mathematically defined in terms of the concept of curve continuity. The big advantage of subdivision surfaces is the ability to create curved surfaces using an arbitrary polygonal mesh as a starting point (also called the *control net*); creating complex curved surfaces with NURBS, for instance, requires stitching adjacent patches together, a tedious process. Subdivision surfaces, like NURBS and other patches, can also be tessellated to an arbitrary number of polygons, yielding a scalable, adaptive representation.

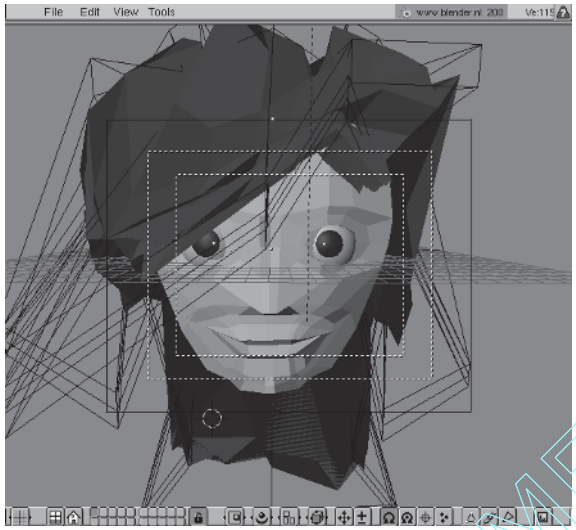


Figure 7-19: A polygonal face rendered without the help of subdivision surfaces. Due to the low polygon count and my current lack of low-polygon modeling skills, the face looks somewhat grotesque.

TEAMFLY

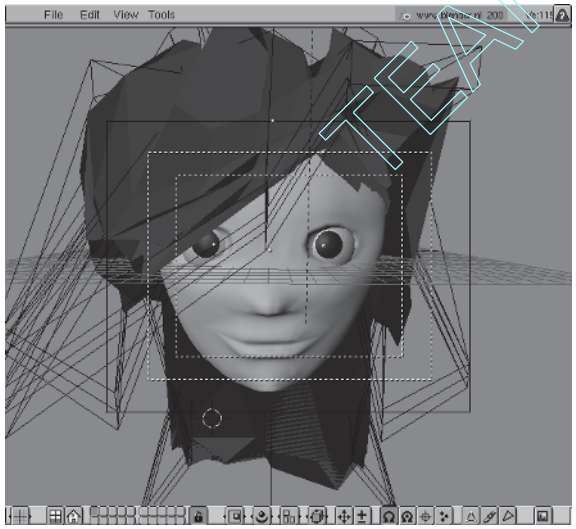


Figure 7-20: The same polygonal face rendered as a subdivision surface. The automatic addition of curvature and smoothness greatly improves the appearance of the model. (The image comes from a planned animation of mine which may or may not see the light of day.)

Blender supports modeling using NURBS surfaces and subdivision surfaces, so it provides a good platform for you to experiment with these curves to see what exactly they are and how they work. To add a NURBS surface, select **Add**, **Surface**, **Surface** from the Toolbox. To change an existing polygon mesh into a subdivision surface, activate the TOGBUT S-Mesh in the EditButtons. But to reap the real benefit of curved surfaces—the scalability and dynamic tessellation—you have to implement the system yourself in your 3D application program. See *Computer Graphics: Principles and Practice* [FOLE92] for a rigorous introduction to the topic, and a few of Brian Sharp's articles for *Game Developer* magazine for more tutorial material with sample programs [SHAR99a] [SHAR99b] [SHAR00].

Team-Fly®

Level of Detail

Level of detail techniques, abbreviated LOD, try to render simpler geometry for more distant objects, and to save more complex geometry for nearby objects. The idea is that with a perspective projection, farther objects occupy fewer pixels on-screen. Therefore, rendering complex geometry with several polygons for a distant object is generally wasted work, because the additional geometric detail all gets projected onto very few pixels, where the additional polygons are likely indiscernible.

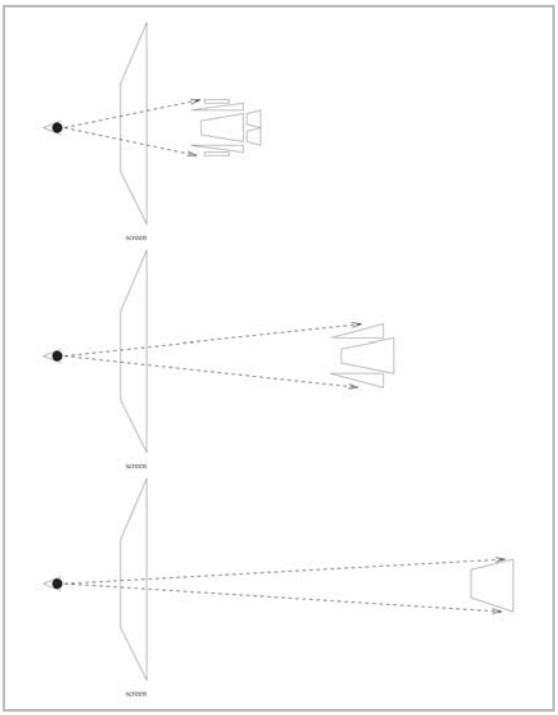


Figure 7-21: The idea behind LOD: farther objects project to fewer pixels on-screen, so we use simpler geometry for farther objects.

LOD techniques generally work in one of two ways: either static or dynamic computation of the LOD models.

First, we can statically create several models ahead of time, with decreasing numbers of polygons. These models can be generated either algorithmically (with one of the methods covered below) or by hand with a 3D modeling program, manually deleting polygons and adjusting the geometry to preserve topology. We store all of these models in memory with each object. Then, at run time, we choose which model to render based on the distance of the object to the viewer. We can use the true radial distance to the camera, or simply use the z object's coordinate in camera space; this is similar to the choice between radial fog and z-based fog.

Second, we can try to dynamically compute the LOD mesh at run time. This is more difficult. Based on the distance to the viewer, we can apply in real time some algorithm to simplify the geometry. Such techniques can be used for landscapes, where a single continuous landscape mesh can be very large and might not lend itself well to partitioning into separate models. In such a case,

we can leave the mesh whole, allowing nearer areas of the landscape mesh to retain their detail, while dynamically simplifying those areas of the mesh farther away from the camera. See the section on landscapes for some more discussion of the technique.

The next few sections discuss some ways of computing different LOD models ahead of time.

Billboards

One of the simplest LOD models is a single polygon. (The only simpler non-empty LOD model is a single pixel.) If the object is suitably far away, a pre-rendered billboard of the object can be used instead of the object geometry. We should store several billboards with the object, each representing the object as seen from a different viewing angle (0 to 360 degrees), and dynamically pick which billboard to use based on the viewing angle. Alternatively, at a slight run-time cost, we can dynamically render the billboard in real time. That is, as soon as we determine that the object is distant enough, we then rasterize it for the current viewpoint into a billboard texture, then render the billboard for subsequent frames. If the viewpoint or the object changes, then the dynamically computed image stored in the billboard texture no longer accurately depicts the geometry; we can either ignore this (because it is far enough away anyway), or recompute the billboard texture. We can also decide only to recompute the texture if the viewpoint has changed by more than a certain amount.

It should be noted that this billboard scheme can be applied not only to single objects, but also to entire areas of a scene [SCHA96] [SHAD96]. Large outdoor scenes with few limits on visibility, such as landscapes or cityscapes, can benefit from such a scheme. This relies on some sort of partitioning of the model, possibly hierarchical, and the association of dynamically computed billboards with each partition (and with each hierarchy node, if hierarchical partitioning is used). Each billboard is a simple rendering of the geometry of the partition which it represents. When rendering, if we encounter a portion of the scene that is too far away, we render it as normal, but save the results of the rendering into the billboard. For successive frames, we then just render the billboard, not the geometry it represents. If the viewpoint changes more than a certain user-definable amount, then the billboard image needs to be recomputed.

With a hierarchical space partitioning of the scene, this means that any part of the hierarchy found to be too far away is simply rendered with a billboard. This is similar to view frustum culling, except with view frustum culling we completely discarded any geometry that was too far away. The idea here is to render the billboard for the distant geometry, which still gives a visual indication of the culled geometry, but is very fast.

Edge Collapse

An algorithmic means of simplifying a triangular mesh relies on *edge collapse* operations [HOPP96]. The algorithm starts with a mesh consisting exclusively of triangles; if other arbitrary polygons are in the mesh, they should first be tessellated to triangles. Then, we make the mesh simpler by moving two separate vertices to one location. By doing this on a closed model, we eliminate one vertex, three edges, and two triangles. We can apply a series of edge collapse operations to reduce a triangular mesh to an arbitrary number of triangles.

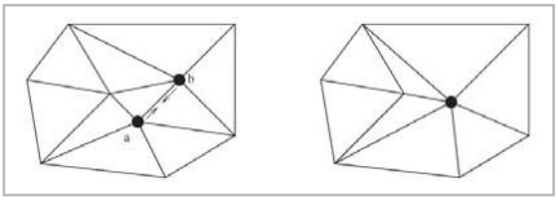


Figure 7-22: A triangular mesh simplified by one edge collapse operation. Vertices a and b are merged into one new vertex.

An edge collapse merges two vertices into one. The important decision to make is the exact location of the new merged vertex. A simple scheme uses the location of one of the original vertices as the new location. This essentially means that one vertex gets merged into the other. The advantage of this scheme is its simplicity and speed; it can even be implemented at run time [MELA98]. The problem is that the approximations generated in this way are not of the best quality. Another method is to search for a new spatial location which most preserves the original shape (and possibly texturing) of the mesh; both vertices then get merged to this new location [HOPP96] [GARL98]. This yields better quality meshes, but requires extra computation time; we must define a cost function for an edge collapse, and find the spatial location minimizing this function.

An interesting property of edge collapsed meshes is that they can be progressively simplified and reconstructed. In other words, each simplified mesh differs very slightly from the previous one, and is constructed based on the previous one. This means that when transforming between LOD models, we can perform a morph from one into the other, providing a smooth transition. Also, we can transmit 3D models in a progressive manner, first transmitting a simple model, and then progressively adding edges. This is useful for display in WWW browsers; the receiver can get a broad impression of the overall model's shape before waiting for the entire model to be received.

BSP Tree

As mentioned in Chapter 5, we can also use a BSP tree, interpreted as a volumetric representation of an object, to perform mesh simplification [NAYL98]. We construct a BSP tree representing the geometry, then prune the BSP tree and create a polygonal model based on the volume described by the pruned BSP tree. Pruning the BSP tree at a shallow level leads to models with fewer polygons; traversing the tree to a deeper level yields models with more polygons.

For this scheme to produce good lower-polygon models, it is important that the tree be constructed such that successively deeper levels of the tree encode more and more detailed features of the geometry, as opposed to a randomly constructed tree that splits the geometry in an arbitrary fashion. The choice of splitting plane is of key importance in the BSP tree algorithm, much as the choice of merged vertex location is important for edge collapses. Bruce Naylor presents cost-based algorithms for evaluating potential splitting planes, and suggests trying both splitting planes which lie on a polygon's face, as well as planes which go through the face vertices and have predetermined directions, such as axis-aligned [NAYL98].

If we ignore the polygonal representation and focus on the tree itself, the tree can also be transmitted in a progressive fashion similar to the edge collapsed meshes. We would transmit the tree nodes from top to bottom, corresponding to a coarse-to-fine bounding of the geometry. The receiver would then need to perform the BSP-to-polyhedron conversion (covered in Chapter 5).

This allows for incremental refinement of the model represented by the BSP tree, since a good BSP tree is a multi-resolution bounding of volume.

Texture LOD Techniques: MIP Mapping

A final type of LOD that deserves mention deals with simplification not of the geometry of an object, but instead of its texture. The reason we wish to do this is to improve the appearance of texture mapped polygons which are viewed from a great distance. In contrast to geometry LOD techniques, which are intended to speed up processing by rendering fewer polygons, this texture LOD scheme mainly aims to improve visual quality. (In some cases this texture LOD scheme can also cause more coherent memory accesses, thus also improving performance, but this is not typically the main goal.)

The problem is that textures viewed at a great distance get mapped onto very few pixels on-screen. With ordinary texture mapping, for each rasterized pixel we only choose exactly one texel from the texture map to display. This causes unwanted moire artifacts, because we are effectively sampling the texture image with too low of a frequency. That is, for a distant texture, several texels should all contribute to the final pixel color on-screen, because several texels all project to the same pixel; but ordinary texture mapping uses just one texel. The plotted color is therefore not representative of the actual light contribution of all texels.

One solution to this problem is to use filtering. This method always dynamically averages the contributions of surrounding texels to arrive at the final pixel color on-screen. This is an excellent solution, but is slow in the absence of hardware support.

Another solution to the problem is to pre-filter the textures into smaller versions, which are used when the polygon is farther away. Then, when performing the texture mapping, we use a texel from the appropriate smaller, pre-filtered texture. We can do this on a per-pixel basis or per-polygon basis. Per-pixel, we compute the distance of each pixel to the camera, select the appropriate texture based on the distance (greater distance causes selection of a smaller texture), then select the appropriate texel from the selected texture. Per-polygon, we simply compute one average distance value for the entire polygon, and then select and apply one entire pre-filtered texture to the polygon, based on the distance. Per-pixel produces better results; per-polygon is faster, but can result in popping artifacts as the entire polygon suddenly shifts between using a smaller and a larger version of the texture.

The smaller, pre-filtered versions of the textures are called *MIP maps*, although the collection of all such MIP maps is also often referred to as the *MIP map*. The idea was introduced by Lance Williams in his article "Pyramidial Parametrics" [WILL83]. The MIP in MIP mapping stands for the Latin *multum in parvo*, meaning many things in a small place, which in this case refers to the storage of the original texture and all of its smaller version in a space 4/3 of the original texture's size. We arrive at this 4/3 factor as follows. Each smaller version of the texture is typically exactly 1/4 of the size of the original image, having exactly half of the horizontal and vertical dimensions of the next larger version. We can compute such smaller versions simply by averaging the colors from a 4 4 region of the larger version. We continue successively reducing the size of the MIP maps until it is just one pixel wide and/or high. Then, the space requirement expressed as a factor

of the original texture size is 1+1/4+1/16+1/64+1/256 and so forth, a series whose limit is 1+1/3, or 4/3.

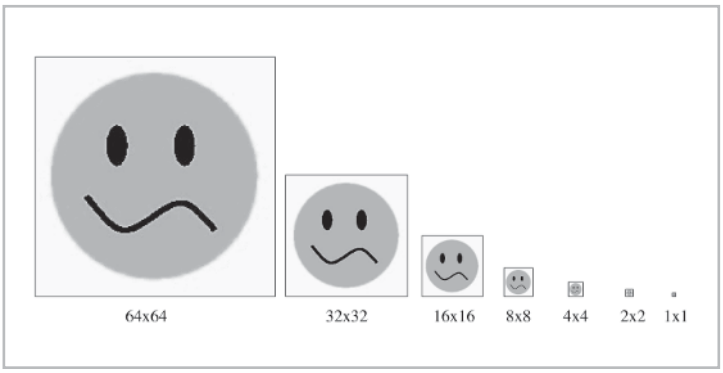


Figure 7-23: A series of MIP maps.



NOTE OpenGL supports MIP mapping; the function `gluBuild2DMipMaps` constructs the smaller mipmaps.

Landscapes

An area of 3D graphics with its own special set of problems and techniques is that of landscapes. This is a unique area due to the storage, generation, and rendering techniques for landscape data. We consider each of these aspects below.

Storing Landscapes as Height Fields

Landscape data is typically stored as a height field, or elevation grid. A *height field* is simply an evenly spaced 2D array of values, where each value represents the height of the landscape at that point on the ground. Height fields are a natural and easy way of storing elevation data, and many natural landscapes can be represented with height fields. Those landscapes that cannot be represented with height fields have cliffs or overhangs, where there would be multiple heights for a particular ground location. Fortunately, many natural landscapes do not have such overhangs, since gravity tends to cause overhangs to eventually fall to the earth, thus forming the simple hills and valleys which we can model with height fields.

We can obtain such height field data for real-world locations based on geographical surveying information. For artificial landscapes, we can create the height field by hand in a normal 2D paint program. If we make a 2D gray scale image, then the darker values of gray could represent lower elevations and lighter shades could represent higher elevations. We can then easily and intuitively paint a landscape, as if we were viewing it from above.

We can use *tiles* to reduce the memory requirements of a large landscape. The idea is to create several small landscape maps, all of which match on the edges. Then, by combining these tiles in various ways, we can generate varied landscapes with reduced storage requirements. Of course, such a scheme introduces repetition into the landscape, but clever arrangement of the tiles can hide this fact. Also, using the same tiles at different elevations is an effective reuse technique. To reuse

a tile at different elevations, the geometry within the tile would need to slope upwards or downwards, so that the tiles match up seamlessly on the edges.

Generating Fractal Landscapes

Height field data for landscapes can also be generated algorithmically. Perhaps the best known class of algorithms for generating such data are the fractal algorithms. Fractal landscape algorithms fundamentally work by starting with a single, undivided polygon. The polygon is then divided into smaller polygons, which adds new vertices. The new vertices are first positioned at the average height of their neighboring vertices, then are additionally randomly displaced up or down. We recursively repeat the subdivision, with the important property that the amount of random displacement gets smaller and smaller as we divide further and further. This means that the first few coarse levels of subdivision establish large-scale hills and valleys; increasing subdivision leads to finer and finer surface details. The exact reason this works is because the random displacements simulate Brownian motion in two dimensions, creating a surface with fractal dimension [PEIT92]. Fractal dimension means that the surface has a fractional dimension between 2D and 3D (hence the term fractal) and appears self-similar. The specific factor by which the amount of displacement is reduced for each recursive subdivision depends on this fractal property, and for square polygons is sqrt($1/2^h$), where h is the level of recursion. See [PEIT92] for the derivation and an extremely broad introduction to the field of fractals and chaos theory.

You can program a fractal landscape generator without too much difficulty based on the previous description. Also, Blender has a fractal subdivision option which you can use to experiment. To try this in Blender, add a plane and remain in EditMode. Then type **w**, and select **Subdivide Fractal** to subdivide all selected vertices in a fractal manner. By repeating this process, you can create more and more fractal landscape detail. Blender, however, only allows you to access and export the data as a polygonal mesh, not as a 2D height field.

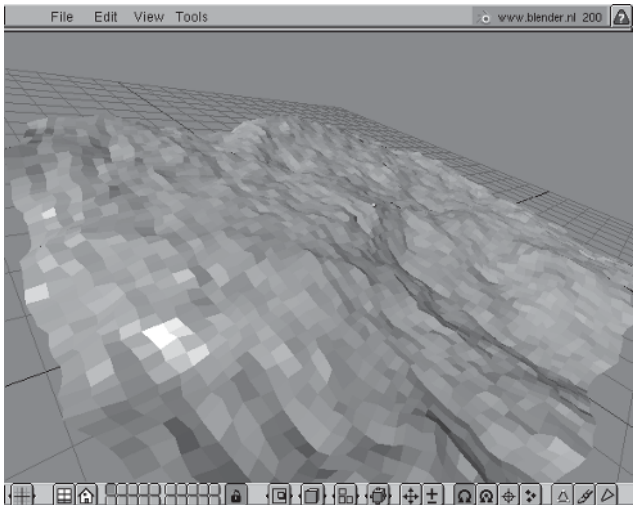


Figure 7-24: A Blender mesh subdivided in a fractal manner.

Rendering and LOD Techniques for Landscapes

LOD algorithms can be of considerable advantage with landscapes because of the large number of polygons needed to draw an entire landscape. The lack of natural partitioning boundaries for landscape meshes makes it difficult to use other LOD or culling techniques which are based on small, separate objects. An extreme example of this is the portal approach described in Chapter 5, which is poorly suited for landscape rendering because of the lack of obvious cell or sector structure in an arbitrary landscape. Thus arises the need for other LOD techniques for landscapes.

Rendering a landscape stored as a height field can be done in a number of ways. One way is to generate curved surface patches which go through the points of the height field, thus creating a smooth landscape which can be dynamically tessellated to the desired level of detail. We could tessellate more for patches closer to the viewer, and fewer for distant patches. However, a critical problem with this approach is to subdivide the patches in a way which avoids cracks between the edges of adjacent patches, since they will be very noticeable. Cracks appear when adjacent edges have different numbers of vertices or differently positioned vertices, so avoiding cracks means ensuring that adjacent edges always line up.

Another way of rendering landscapes is to create triangles between alternate rows of the height field, as we saw with parametric bicubic surfaces, and to render these triangles. Instead of always rendering all of the triangles, we can try to use a dynamic LOD scheme to reduce the number of rendered polygons in distant areas. The ROAM algorithm is one such algorithm. ROAM stands for Real-time Optimally Adapting Meshes [DUCH97]. This algorithm relies in a special data structure, the triangle bintree, which allows for run-time, view-dependent alteration and simplification of parts of the landscape mesh based on their distance from the camera.

Finally, it is also worth mentioning that for small landscapes with relatively few polygons, and fast enough graphics acceleration hardware, it is sometimes possible to completely ignore LOD issues and simply render all triangles within the landscape for every frame. This could conceivably be used in a mixed indoor/outdoor engine, where the outdoor parts of the world are isolated small landscapes. The isolated small landscapes could then be rendered in their entirety. If we combine this with a portal scheme and place the landscape areas in their own sectors, the landscape regions do not degrade rendering performance when they are completely blocked by intervening walls of an indoor region.



NOTE One older way of rendering landscapes relies on rendering the height field as a series of vertical strips in front-to-back order, and restricting camera movement such that rotation around the VFW vector is forbidden; in other words, the VUP vector always points straight up. By restricting camera movement in this way, the projection of the height field to the screen always results in a series of perfectly vertical strips. By rendering these front to back, we can cull large regions of the landscape which are obscured by nearer, higher vertical strips. We can only see parts of the landscape to which the line of sight is not blocked by the nearer strips, meaning that an intersection calculation from the eye through the top of the highest foreground strip determines which is the next visible part of the background. Due to the restriction on camera movement, and the lack of hardware acceleration for this method, this scheme has lost popularity in recent years, though the scheme does provide an interesting alternative.

Camera Tracking

Another advanced technique useful for interactive 3D graphics applications is camera tracking. By camera tracking, we mean that the camera should always follow a particular subject. This is particularly useful for third-person games, where the player controls the movement of a character throughout the virtual world. The world is seen not through the character's eyes, but instead from the viewpoint of an external camera following the movement of the character. In such an application, it is important that the camera always keep the target in sight.

An appropriate rotation matrix can ensure that the camera always points towards the target. We take a vector from the camera to the target, normalize it, and use this as the camera's VFW vector. We cross the vector (0,1,0) with the VFW vector, normalize the result, and use this as the VRI vector. Then we cross the VFW and VRI vectors to obtain the VUP vector. This ensures that the camera looks at the target and is oriented vertically. We can also compute the distance between the camera and the target, and ensure that the camera never gets too close to or too far away from the target by moving the camera whenever the distance falls outside of the tolerance. A narrow tolerance range causes the camera to follow the target at an almost constant distance, as if it were attached with a wire to the target, whereas a wide tolerance allows the camera to move into place, then rest and follow the action just by looking, until the target moves significantly farther away.

One problem with this simple approach is that it is possible that the camera is looking toward the subject, but that one or more intervening polygons actually obstruct the line of sight. There are several solutions to this problem. One is to determine which polygons intersect the line of sight (using a line-polygon intersection test; see Chapter 8), and to refrain from drawing these polygons, or to draw them with slight transparency (using an alpha channel). This produces a sort of "cutaway" view of the 3D model allowing the target to be seen.

A more complex solution involves finding a new spatial position from which the target can be seen—that is, a spatial position where the line of sight to the target is not blocked by any intervening polygon. One way of doing this, suggested to me by Ton Roosendaal [ROOS99], is to start at the target position, and trace a ray back to the camera. If the ray hits any polygon, the line of sight is blocked, but we at least know that the location of the first polygon intersection is a spatial location from which the target can certainly be seen. It is not the only such location, but it is definitely one such location. Therefore, the goal is to move the camera to this location. If the camera does not collide with the environment, then we can simply move the camera to the new location, allowing the camera to sail unhindered through the obstructing geometry to its new vantage point. If the camera is not allowed to move through walls, then we must find a path around the obstructing geometry. We can do this by repeatedly reflecting the visibility vector off of the obstructing geometry, thereby incrementally tracing a path backwards from the target to the camera; then, the camera traverses this path forwards toward the target.

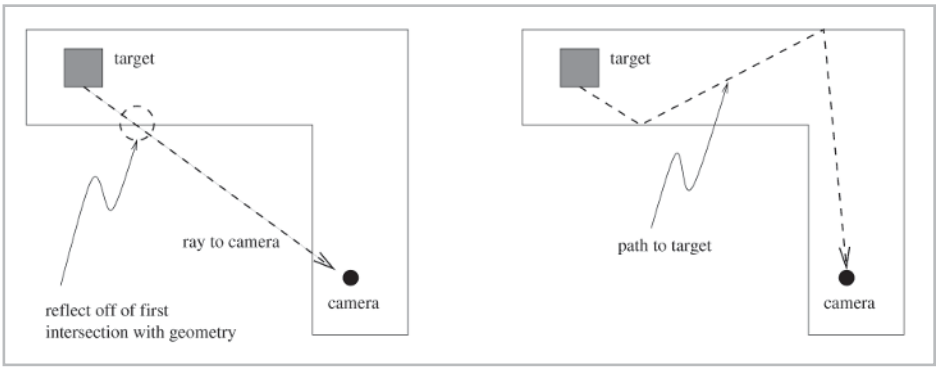


Figure 7-25: Roosendaal's method for camera tracking. The line of sight from the camera to the target is blocked, so we trace a reverse ray from the target to the camera, bouncing the ray off of any intervening geometry, until we reach the camera. This is then the path the camera must take through the geometry to see the target.

The path thus determined does lead to the target, but in a fairly inefficient way. We can improve this by continually checking for target visibility while the camera is moving along the path. If the target suddenly becomes visible, then we can stop tracing the path; we found an earlier solution along the way. Otherwise, we simply trace the path all the way to its end, which eventually leads to a point where the target is certainly visible.

The World Foundry game development kit [SEGH97] adds more designer control to the camera at run time. World Foundry introduces the concept of a *director* object, which can then select among various *camera shots*. Each camera shot has a different focus of attention, and the director uses some user-defined criteria, specified in a script file, to choose among the camera shots. This allows for a great deal of artistic freedom with camera movement.

Well-executed camera movement can have a significant impact on the effectiveness of an animation sequence. Camera control in 3D environments is similar to camera control in films, where cinematographers have developed a visual language in which camera placement plays an important role. The article "Declarative Camera Control for Automatic Cinematography" in *Proceedings of AAAI '96* explores some of the elements of cinematography and develops a high-level declarative language for specifying computerized camera shots in director's terms, along with a compiler and heuristic evaluation system which control camera movement within the 3D environment [CHRI96].

Summary

In this chapter, we took a quick look at some important graphical techniques useful for interactive 3D programs. In particular, we looked at:

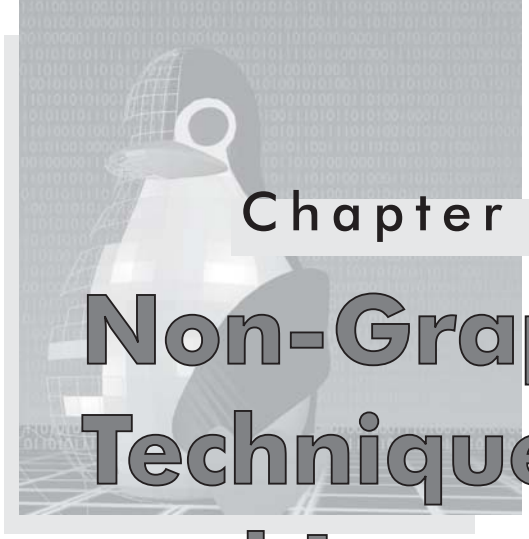
- Special effects techniques: environment mapping, billboards, lens flare, particle systems, bump mapping, and multi-pass techniques

- Natural phenomena: water, smoke, fire, and clouds
- Advanced techniques: curved surfaces, level of detail techniques, landscapes, and camera tracking

Combining the techniques of this chapter with the ideas and code from the rest of the book allows us to create realistic, interactive Linux 3D applications.

Although our applications can now achieve a degree of visual realism, interactive 3D environments require more than just visuals. Other forms of interaction with the environment are also desirable. The next chapter discusses non-graphical techniques which are very often used by 3D graphics programs to increase the illusion of reality on levels other than visual.

TEAMFLY



Chapter 8

Non-Graphical Techniques for Games and Interactive Environments

We've almost reached the end of this book. Until now, the content of this book has, not surprisingly, focused on the graphics techniques necessary for creating visually appealing and realistic 3D applications. But in such 3D applications, often visual quality is not the only important factor. 3D applications are often simulations of reality, such as games, virtual reality, and so forth. In such programs, the overall effect of the application is a combination of visual and other cues provided by the program. This chapter is devoted to these other cues. By devoting some attention to these other issues, we can produce 3D programs that not only look but also sound and act realistically.

In this chapter we look at the following topics:

- Collision detection
- Physics
- Artificial intelligence
- Digital sound
- Networking

Sound

Though this is a book on 3D graphics, writing our programs to involve more senses than just sight often enhances the entire experience. The next easiest sense to integrate is the sense of sound.



NOTE The sense of touch has already been implemented in *haptic* systems, where force feedback gives the impression of actually coming in contact with the virtual objects. Haptic devices can involve just one finger, an entire hand, or even the entire body. The sampling rate for haptic systems lies far above that of graphics systems; rates must lie in the thousands of samples per second to allow the user to physically interact in real time with the virtual objects and to avoid the feeling of vibration in the feedback device. The senses of smell and taste are more difficult to integrate reliably into virtual environments; I am not aware of any non-experimental products in this area.

Digital audio is surprisingly simple to implement under Linux, or under Unix systems in general. For comparison, I'd like to digress for a moment to compare the Linux audio interface with that of DOS. Years ago, I wrote a music editor for digital music files under DOS, called ModEdit. Accessing the sound card under DOS was an incredibly tedious affair (and in the end was done by my programming partner Mark Cox), involving interrupts, assembly language programming, precise timing, and direct access to hardware I/O ports. I recall looking at sound card access code in assembler: it was not a pretty sight. Then a while later, I happened to find a MOD file player for SunOS, a Unix variant. It was written entirely in C, and the audio output consisted of two lines: a write command to the special file `/dev/audio`, and a `usleep` call to wait the appropriate number of microseconds before sending the next sample to the audio device. Digital audio couldn't be simpler; the Unix philosophy of everything is a file also applies to device drivers, including those for the sound card. So, under Linux, we just need to write to `/dev/audio` to create a sound.

Let's first look at the basics of digital audio with a simple example program that directly accesses the audio device to create a sound. Then, we'll look at a more convenient way of playing sounds, including background music, CDs, and multiple simultaneous sounds, using the RPlay server.

Basics of Digital Sound

Humans physically perceive sound because vibration of matter causes the vibration of surrounding air particles and the creation of longitudinal sound waves. These sound waves propagate through the air and cause a corresponding vibration in the ear drum, where the movement is then transmitted to the brain and interpreted as sound. Loudspeakers can reproduce other sounds because they vibrate in such as way as to produce the same sound waves created by the original source.

A loudspeaker is nothing more than an electromagnet connected to a large cone. Variations in current flowing through the electromagnet's coils cause a movement of the electromagnet itself, which correspondingly causes a movement in the cone. The cone's movement, in turn, causes the vibration of the air and the reproduction of the sound. Therefore, a loudspeaker converts electrical current levels into physical air vibrations.

Digital sound works by controlling a speaker's electromagnet digitally. We send an instruction to the sound card hardware (through an I/O port or an appropriate device driver) to play a particular digital sound value, typically a signed integer. The sound hardware interprets this number and converts it to a specific voltage and current level on the Speaker-Out jack, which then ultimately flows through the loudspeaker coils and creates the sound. Therefore, a single integer

number controls the loudspeaker displacement. By very quickly sending a large number of varying numbers to the sound hardware, we can reproduce sounds with acceptable aural fidelity. CD audio uses a sampling rate of 44,100 hertz, meaning that 44,100 digital samples per second are sent to the speaker. Due to an artifact of discrete sampling, the highest frequency that can be reproduced is only half of the sampling rate, meaning CDs can reproduce sounds up to 22,050 hertz. Seen the other way around, the minimum sampling rate (called the Nyquist rate) necessary to faithfully reproduce a signal with frequency *f* is twice this frequency, 2*f*.



NOTE Since fundamentally digital audio works by converting an integer number into a physical voltage level, it is actually quite possible to build your own sound card for digital audio purposes. The idea is to output digital sound values to the PC's parallel port; in this case, each 8-bit byte which is sent to the parallel port gets neatly mapped to eight separate pins on the parallel port. Then, with a series of resistors connected in a ladder-like fashion (called a D/A converter, for digital-analog), we can convert the electrical signals from the eight pins into one single combined voltage level. This voltage level can then be fed into an amplifier, where the small voltage variations get amplified to a level enough to move a loudspeaker cone and reproduce the audio signal. Making such D/A converters was common practice in the early 1990s, when PC digital audio was still fairly young. My first sound card was such a home-brew D/A converter, which I built for use with ModEdit. Since almost every PC nowadays comes with a sound card, you generally don't need to build your own sound card to enjoy digital audio, but it's useful to know that there's no magic involved—just a bit of solder.

So, under Linux, how do we write values to the sound hardware? As we hinted at earlier, the key is the special device file `/dev/audio`. We simply open this file, write numbers to the file, and hear the output. `/dev/audio` is not actually a file, but is rather a *character special device*. We can access it just like a file, which makes programming very straightforward, but in reality, writing and reading characters to this file actually invokes a special driver, which then does something particular with the data. All files in `/dev` are such special files; the `mknod` program is used to create these files. Each special file has a *major number* and a *minor number*, specified by `mknod`, which allows the operating system to identify the proper driver to load when accessing that file. In the case of `/dev/audio`, the major number is 14, and the minor number 4, as you can verify with an `ls -l` command.

Two items determine the sound which is produced when writing to `/dev/audio`: which numbers we write and how fast we write them. The following sample program, `square`, produces a so-called *square wave* by alternately writing a low value, then a high value to the audio device, omitting any values in between. This produces a somewhat mechanical sounding tone. The sample program writes the numbers as fast as it can to the sound device, meaning that the default sampling rate set by the audio driver determines the actual frequency heard. The `pat` array controls the values written to the sound card. Experiment with these values; for instance, if you add additional leading zero values, the resulting tone decreases in pitch, because the square wave signal bursts become separated by a larger interval, leading to lower frequency.



NOTE The sample program `square` does not use and is not part of the l3d library, and is very Linux-specific. Therefore, in contrast to most of the other sample programs, its executable file is compiled directly into the directory containing the source code files. The program is in `$L3D/source/app/audio`.

The actual interpretation of the values written to the sound card depends on the sound card driver and hardware. Sometimes, signed byte values are used, which means that a value of zero represents no displacement of the speaker cone, a value of 127 represents the maximum positive displacement, and –128 represents the maximum negative displacement. Alternatively, unsigned values can be used, meaning that a value of 128 represents no displacement, 0 represents negative displacement, and 255 represents positive displacement. Furthermore, we also often use two bytes (16 bits) instead of just one byte (8 bits). To set these parameters yourself in the sound driver, you can use the `ioctl` function, which is a general routine that sets parameters for various device files. For the specific parameters allowable for the sound file `/dev/audio`, see file `/usr/src/linux/include/soundcard.h`, and search for the string `ioctl`.



NOTE Note how the open source philosophy allows and encourages programmers to literally go straight to the source (code) to understand exactly which parameters are allowable. There's no danger here that the documentation is out of date since you are looking at the actual code executed by the operating system.

Listing 8-1: `square.cc`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

main() {
  FILE *fp;

  fp=fopen("/dev/audio","wb");
  char pat[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                128 , 128, 128, 128, 128};
  char i = 0;
  int l = sizeof(pat);
  while(1) {
    fputc(pat[i++], fp);
    if(i>=l) i = 0;
  }
}
```

This program thus shows us how to play one sound to the sound card: we write sound data directly to the audio device in a linear fashion. Digital sound data in this format is said to be in *raw* format. The Sox program can convert sound files to and from various formats, including raw files. This means that most sound files can be converted to raw format, at which point we can read the data directly into memory, then dump it directly to the audio device (possibly first setting the sound card parameters with `ioctl`, as discussed above).

To play multiple sounds simultaneously, each sample we send to the sound card is simply the sum of all integer signal contributions from all sources, clamped to lie within the maximum and minimum allowable range. Importantly, we do not average the signals together! In other words, we do not add all the signals, then divide by the number of signals present; we only add all the

signals together. Imagine the simple case of a loud sound and a completely silent sample being played together. If we averaged the two, then the loud sound would decrease in amplitude by one half, which makes no physical sense: a sound does not decrease in amplitude just because another sound (especially a silent one) happens to be playing at the same time. Instead, since sounds can be viewed as waves varying over time, we simply add all contributions of all waves together to get the total sound wave sample at that point in time. It is possible that all sounds together exceed the maximum or minimum allowable sound values, which results in clipping of the audio signal; this is unavoidable, since the combined signal is then simply too loud to be correctly represented by the audio device. But on the average, with sound waves that have both positive and negative digital values, we can expect that their summed contribution will normally lie within bounds; when one signal is at a high, it's very possible the other signal is at a low, resulting in a medium-level summed contribution. The moral of the story: just add multiple signal contributions together to play several sounds simultaneously.

While this shows us how to play sounds, there is a lot more to actually integrating sound support into our programs. Ideally, we would like direct support for common sound formats, a queue which allows us to submit sound requests, and an event notification mechanism which tells us when sounds finish playing. Fortunately, there is an excellent, easy-to-use sound server for Linux, the RPlay server.

The RPlay Server

RPlay is a flexible network sound server which allows sounds to be played to and from local and remote Unix systems. It can play multiple simultaneous sounds with differing volumes, understands common formats (au, wav, voc, raw, and many others), can be extended to play arbitrary sound formats through helper applications, supports audio CD tracks, has an event notification mechanism, and runs parallel to the application program. Since RPlay is a separate server program, which incidentally can run on a separate computer from the one running the application, playing sounds requires very little work on the part of the client program. We merely need to send a command to the server, specifying the sound file we want to play. The rest is handled by the server. The command is sent over a TCP/IP socket, as we see shortly. (TCP/IP stands for Transmission Control Protocol/Internet Protocol; TCP is a protocol built on top of IP.)

After installing the RPlay server (see the Appendix), start the server as follows: type **rplayd** and press **Enter**. This command starts an RPlay server on the same machine, listening on network port number 5556. This means that the RPlay server can accept an incoming IP connection on this port. A *port* is simply a number under which a service of a machine on a network can be accessed. So, any machine which can initiate a network connection to port 5556 can access the RPlay server. In our case, our application program and the RPlay server both run on the same machine, meaning that our application program must open a connection to the RPlay server.

Let's next discuss how a sound client connects to the RPlay sound server via TCP/IP networking.

Using TCP/IP Networking to Communicate with the Server

Opening a connection to the RPlay server is done by creating a *socket*, which is an endpoint for network communication. A socket is similar to a file descriptor in C; we conceptually open it, write to or read from it, and close it. In this sense, a socket functions similar to a file, except the data comes from the network. This is similar in spirit to character special files, which look like files but which actually send and receive data to and from devices.

We create a socket with the `socket` system call, then call `connect` to connect the socket with the RPlay server at address 127.0.0.1 and port 5556. The Internet address 127.0.0.1 is a special address which always represents the local host. After opening the socket, we can use the `send` function to send data to the socket. In our case, we don't need to worry about receiving data from the server, so we only need to write to the socket.

The data we write to the socket to control the RPlay server are simple ASCII strings. RPlay supports a number of commands; see the RPlay online documentation for a comprehensive list. The only command we are interested in for now is the command to play a sound file. The simplest form of this command looks as follows:

```
play count=1 volume=64 sound=crash.au
```

The `count` parameter specifies how many times the specified sound should be played. The `volume` parameter specifies the volume, with 0 being the lowest and 255 being the highest. The `sound` parameter is the full path to the sound file to be played. The sound file must be accessible to the RPlay server, which is why it is advisable to specify a full path name. Alternatively, a path name relative to the directory in which the server was started may be specified. Yet another possibility would be to stream the actual sound data over the socket to the server.

The next sample program `playtest` creates a socket and connects it to the RPlay server. Run the program as follows. First, change to the program's source code directory `$L3D/source/app/audio`. Compile the program with **make**. Next, kill any previous running copies of the RPlay server: type **killall -9 rplayd** and press **Enter**. Then, start the RPlay server from this directory: type **rplayd** and press **Enter**. (We start the server in this directory to allow the sample program to specify just the filename of the desired file, without the full pathname.) Next, start the sample program: type **playtest** and press **Enter**. The program waits for you to press **Enter**; each time you do, a command is sent via a socket to the RPlay server to play the file `crash.au`, located in the same directory.

Listing 8-2: `playtest.cc`

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>

main() {
  int mysocket;
```

```
  struct sockaddr_in addr;
  struct sockaddr_in boundaddr;

  addr.sin_family = AF_INET;
  addr.sin_port = htons(5556);
  inet_aton("127.0.0.1", &addr.sin_addr);
  mysocket = socket(PF_INET, SOCK_STREAM, getprotobyname("IP")->p_proto);
  connect(mysocket, (sockaddr *)&addr, sizeof(addr));

  while(1) {
    char s[256];
    gets(s);
    char cmd[] = "play count=1 volume=64 sound=crash.au";
    printf("sending following command to server: %s",cmd);
    send(mysocket, cmd, strlen(cmd),0);
  }
  shutdown(mysocket,2);
}
```

Notice that the sound plays as soon as you press Enter, and that the client program is very simple indeed. We simply send the appropriate command to the server. This is the advantage of using an external sound server.

If you don't hear any sound when running this program, then make sure that you started the RPlay server from the directory containing the file `crash.au`. You can also test to make sure that the server is running as follows: type **telnet 127.0.0.1 5556** and press **Enter**. This opens up a telnet connection to the RPlay server. You should see the RPlay welcome message, and can even issue RPlay commands directly. Press **Ctrl+], q** to quit the telnet session. If you are unable to establish a connection with telnet, then the RPlay server is not running. Other possibilities to check are that the volume on the sound card—both in software and in hardware—has been set loud enough, and that your sound card fundamentally works properly under Linux. The Sound HOWTO, usually located in `/usr/doc/howto`, explains how to get your sound card set up properly.

This sample program shows how easy it is to add audio to our programs by sending simple commands over a socket to the RPlay server on port 5556. Let's now encapsulate this functionality into l3d classes.

Class l3d_sound_client

Class `l3d_sound_client` represents a user of a sound server. It is an abstract class that allows us to access sound services in l3d programs without needing to directly hard-code in Rplay-specific code. This class is designed to be abstract enough so that other sound clients, not based on RPlay, can also inherit its interface.

Listing 8-3: `soundclient.h`

```
#ifndef __SOUNDCLIENT_H
#define __SOUNDCLIENT_H

class l3d_sound_client {
  public:
    virtual void send_command(const char *cmd) = 0;
    virtual void connect(const char *addr) = 0;
    virtual ~l3d_sound_client(void) {};
```

```
};

class l3d_sound_client_factory {
  public:
    virtual l3d_sound_client *create(void) = 0;
};

#endif
```

Abstract method `connect` connects to a sound server. Parameter `addr` takes a character string representing the address of the server to connect to.

Abstract method `send_command` sends a character string to the sound server.

Note that this interface is overly simplified; a more useful interface would abstract useful operations into virtual methods, such as `play_sound`, `stop_sound`, `get_sound_status`, and so forth. The current approach has an interface with strong implicit dependencies on structured ASCII messages. To preserve compatibility, any new sound client subclasses must not only conform to the explicit class declarations, but also to the implicit ASCII message protocol. Therefore, this class is mostly illustrative in nature, though its structure does provide some minimal flexibility for the future.

Class l3d_sound_server_rplay

Class `l3d_sound_client_rplay` is a sound client which can connect to an RPlay server.

Listing 8-4: `scli_rplay.h`

```
#ifndef __SCLI_RPLAY_H
#define __SCLI_RPLAY_H

#include "soundclient.h"

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

class l3d_sound_client_rplay_factory : public l3d_sound_client_factory {
  public:
    l3d_sound_client *create(void);
};

class l3d_sound_client_rplay : public l3d_sound_client {
  public:
    virtual ~l3d_sound_client_rplay(void);
    void connect(const char *connect_string);
    void send_command(const char *cmd);
  protected:
    int sound_socket;
    int connected;
    struct sockaddr_in addr;
};

#endif
```

Listing 8-5: `scli_rplay.cc`

```
#include "scli_rplay.h"

#include <sys/socket.h>
```

```
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

l3d_sound_client* l3d_sound_client_rplay_factory::create(void) {
  return new l3d_sound_client_rplay;
}

void l3d_sound_client_rplay::connect(const char *connect_string)
{
  char parms[1024];
  char server_ip[256];
  char server_port[16];

  strncpy(parms,connect_string,sizeof(parms));
  parms[sizeof(parms)-1]=0;

  char *param;
  param = strtok(parms," ");
  strncpy(server_ip, param, sizeof(server_ip));
  server_ip[sizeof(server_ip)-1]=0;

  param = strtok(NULL," ");
  strncpy(server_port, param, sizeof(server_port));
  server_port[sizeof(server_port)-1]=0;

  addr.sin_family = AF_INET;
  addr.sin_port = htons(atoi(server_port));
  inet_aton(server_ip, &addr.sin_addr);
  sound_socket = socket(PF_INET, SOCK_STREAM, getprotobyname("IP")->p_proto);
  connected = ( ::connect(sound_socket, (sockaddr *)&addr, sizeof(addr)) == 0 );
}

void l3d_sound_client_rplay::send_command(const char *cmd) {
  if ( !connected || send(sound_socket, cmd, strlen(cmd),0) <= 0 ) {
    //- apparently there was an error. try, but just once, to re-establish
    //- contact to server.

    connected = ( ::connect(sound_socket, (sockaddr *)&addr, sizeof(addr)) == 0);
    if(connected) {
      send(sound_socket, cmd, strlen(cmd),0);
    }
  };
}

l3d_sound_client_rplay::~l3d_sound_client_rplay(void) {
  shutdown(sound_socket,2);
}
```

Overridden method `connect` interprets the address as being a string containing the server's IP address followed by the port number of the RPlay server. It opens a socket to the appropriate host and port.

Overridden method `send_command` sends the ASCII string, passed in parameter `cmd`, directly over the socket to the sound server, allowing the server to execute the command. As mentioned before, this has the undesirable side effect of introducing implicit dependencies on the ASCII RPlay protocol. Though our l3d programs won't access this class directly, instead going through the abstract class `l3d_sound_server`, the fact that the ASCII command strings (which our l3d programs do specify directly) are based on the RPlay protocol causes the application code to be dependent on this protocol in a way which is not obvious just by looking at the abstract class declarations. But, since this chapter is largely a do-it-yourself chapter anyway, it's your task to see if you can design a better interface. Doing so would probably require researching several sound servers to see what a truly common command set would be, then implementing this command set as a set of pure virtual functions overridden by concrete subclasses. Remember that finding appropriate abstract classes is generally the hardest part of object-oriented design. This is a good example; to understand the abstract properties of the problem domain, you have to look at enough representative examples (or develop a general enough command set through sheer analytic reasoning, probably an impossible task).

TCP/IP Networking

Communicating with the RPlay server required us to open a socket to the RPlay server to send data to the server via TCP/IP. In general, it can be useful for 3D programs to communicate with other programs or with each other via TCP/IP sockets. This can be used for synchronizing multi-user environments in TCP/IP networks, allowing events from one user's machine to propagate to another user's machine. So, let's look at the general way in which we can achieve TCP/IP networking between two programs.

As we saw with RPlay, servers make their services available via an IP address and a port. This means that any two machines that are connected via a TCP/IP network can communicate with one another using the techniques discussed below. This includes machines inside of a LAN (local area network), as well as machines literally located on opposite sides of the Earth connected via the Internet. This is one aspect that makes networking so exciting. Linux support for networking is extremely good; after all, the Internet was born on Unix machines. With just a few lines of code you can write an Internet server and client which enable worldwide communication.

There are two parts to any network connection: a client and a server. The same program may simultaneously be a client and a server. In a client role, a program establishes connections to other machines' services using their IP/port combinations. In a server role, a program listens for connections on a particular port, and answers any incoming requests on that port.

The following sections show programs for both an IP client and server. The programs are located in directory `$L3D/source/app/network`.

The Client

Sample program `client` is a network client that attempts to connect to a server at a specific IP address and port number. It then allows you to enter any input on a line, which is then sent to the

Team-Fly®

server. The server's response to your input is then displayed. This continues indefinitely. This model of communication is quite general: any information is sent to the server; the server does something with the information; and then the server returns some other information.

This program is very similar to the RPlay client we saw earlier. The only real change is that it displays the server's responses to our messages. Let's now discuss the program structure in a bit more detail.

The address of the server is specified in a C structure of type `sockaddr_in`, where the `in` stands for Internet. We set the `sin_family` of this address structure to be `AF_INET`, the address family for Internet addresses. The `sin_port` member of the address structure specifies the port number to connect to. We must specify the port number using function `htons`, which changes the byte ordering of the specified number from host to network order, explaining the "h" and the "n" in `htons`; the "s" denotes the data type, a short integer. The `sin_addr` member of the address structure contains the actual Internet address of the machine to connect to. We set this member using the `inet_aton` function, which converts an IP address in dotted quad notation (such as 127.0.0.1) to binary data in network byte order.

We then create a socket with the function call `socket`, specifying the following parameters: `PF_INET` to specify the Internet Protocol family, `SOCK_STREAM` to specify a sequenced and reliable connection stream (as opposed to unreliable or non-sequenced messages such as ping requests), and `getprotobyname("IP")` to specify the Internet Protocol for communication over the socket.

The `connect` function then connects the given Internet socket with the Internet address and port we previously stored in the address structure. The `send` and `recv` functions send or receive data from the socket, using a buffer to facilitate the transfer of data. These functions normally return the number of characters processed. If we notice that the number of characters processed at any stage is zero or negative, then some error occurred, and we shut down the socket with `shutdown`.

Listing 8-6: `client.cc`

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <sys/time.h>

main() {
  int mysocket;

  struct sockaddr_in addr;
  struct sockaddr_in boundaddr;

  addr.sin_family = AF_INET;
  addr.sin_port = htons(5559);
  inet_aton("127.0.0.1", &addr.sin_addr);
  mysocket = socket(PF_INET, SOCK_STREAM, getprotobyname("IP")->p_proto);
  if(connect(mysocket, (sockaddr *)&addr, sizeof(addr))) {
```

```
    printf("connect to server failed");
    exit(-1);
  };

  int ok = 1;
  while(ok) {
    char response[1024];
    int response_len;

    response_len = recv(mysocket, response, sizeof(response),0);
    response[response_len] = 0;
    printf("server said: %s",response);

    char s[256];
    gets(s);
    printf("sending following string to server: %s",s);
    ok = ( send(mysocket, s, strlen(s),0) > 0 );
  }
  shutdown(mysocket,2);
}
```

In the sample program, we hard-coded the IP address and port number of the server. You can change these values to any valid Internet address and port number to connect to any machine reachable via the Internet Protocol. A server should be listening on the specified IP address and port number. For the purposes of this section, we are interested in creating our own server, so that our 3D programs can exchange data with one another in real time. Writing a server is our next topic.

The Server

Writing a simple IP server is not much more difficult than writing the client. We first declare an Internet address structure of type `sockaddr_in`, and fill it with the IP address and port of the *server*. In other words, the server fills the address structure with its own IP address, and with the port number on which it will be listening. In the sample program `server`, we specified address 127.0.0.1 for the local host, but in a real application with separate client and server machines, you would replace this with the actual IP address of the server machine in the network. The port number specified in the server should be the same as the port number in the client; otherwise no communication will be possible.

Note that one machine can have several IP addresses; indeed, an IP address is not assigned to a "machine," but is instead assigned to a *network interface*, whereby one machine can have several network interfaces (such as network cards, and the PPP modem interface). To allow incoming connections on any of the server machine's IP addresses, we can use the assignment `addr.sin_addr=htnol(INADDR_ANY)`, where `INADDR_ANY` means that any IP address on any of the server's network interfaces will accept an incoming connection.

You obtain your IP address from your Internet service provider (ISP). If you use a dial-up connection to the Internet, then your IP address is probably dynamically assigned each time you connect to the Internet; if you have a cable modem, you probably have a static IP address which always stays the same. Putting servers on dynamic IP addresses requires the server to somehow announce its location to potential clients; one way is to have a WWW site with a static IP address,

containing a page with the current dynamic IP address of the server. Whenever the server machine connects or disconnects from the Internet, it updates this WWW page.

Within a local area network (LAN) that never connects to the Internet, you can choose to assign your own arbitrary IP addresses to the machines in the network, since communication only occurs among the local computers. Under Linux, you configure the IP address with the command `ifconfig`, which allows you to configure a network interface, including specification of the interface's IP address. Typically, LAN IP addresses adhere to the convention specified in RFC1597, where IP addresses lie within one of the following ranges: 10.0.0.0 through 10.255.255.255, 172.16.0.0 through 172.31.255.255, or 192.168.0.0 through 192.168.255.255. These IP addresses are guaranteed never to be assigned to computers actually connected to the Internet, meaning that they are always free for local use. Of course, if you are using a LAN IP address as opposed to a real IP address from an ISP, then your clients and servers can only communicate with other machines in the local network, not with the Internet at large. For more information on Linux and networks, see the Linux Networking HOWTO.

It's also worth mentioning how you choose a port number for your server. Only one service can occupy a particular port on a particular machine. Several port numbers are assigned to standard services and should not be used by your server; if you do, then either you will block the intended server when it tries to start, or the other server will have already started and will prevent your server from starting. The file `/etc/services` documents some of the more common port numbers which are already taken. Historically, the lower port numbers are reserved for the system, and higher ones for user programs, but there are now so many Internet server programs which can provide useful services that it's difficult to know if your port number might not possibly also be used by some other program, sometime in the future. The guideline is generally to choose a high port number (in the thousands or ten thousands) which is not listed in `/etc/services`, and hope for the best. For temporary servers on machines over which we have complete administrative control, we are more or less free to choose any port number, since we can then simply kill any other servers that want to use that port. You can get a list of all currently active ports with the following command: **netstat -a -n**.

Let's now return to the flow of control for a typical server program. We just discussed how to create a socket and assign it an IP address and port number. After creating the socket, we call `bind` on it to bind it locally to the specified address. Contrast this with the client case: the client called `connect` to establish a remote connection; the server calls `bind` to create a local reception point for a connection.

Next, the server calls `listen` on the socket to wait for incoming connections. The first parameter is the socket; the second parameter is a backlog parameter specifying how many simultaneous pending connections may exist.

Then, we enter an infinite loop. The first command in the loop is `accept`. The `accept` command accepts an incoming connection on the socket. If no connection is pending, the command *blocks* until a connection arrives; this means that the program code does not continue execution until a connection is made from the outside.

As soon as a connection arrives, the `accept` command allows execution to continue. `accept` returns a new socket, which is the new communication channel with the just-connected

client. This means that immediately after connecting to the server port (in this example, 5559), the client is actually immediately redirected to another free port (for instance, 55237). This process happens automatically through the accept call. The redirection is needed because if the communication with the client took place on the original port, no other clients could connect. This means that the server port is exclusively for handling incoming connections; ongoing "conversations" with clients take place after connection on a separate socket, created and returned by the accept call.

After accepting the connection on a new socket, we have an interesting quandary that the server must now respond to any queries which the newly connected client made, but it must also continue to monitor and accept connections on the original port. As more and more clients connect, the server would have more and more work to manage. The standard way of handling this is to spawn a new process at the operating system level to handle the newly connected client, while the original server process continues to handle incoming connections on the server port. The Linux command to spawn a new process is `fork`. The `fork` function creates an almost identical copy of a process, called the *child process*, and both copies continue execution after the `fork` command. The original process, called the *parent process*, differs from the child process in only one way: the return value for `fork` in the parent is non-zero and represents the process ID number of the child process; the return value for `fork` in the child process is always zero. Therefore, we call `fork` and check the return value. If it is non-zero, then we are the parent process, and continue waiting for connections with the `accept` function. If it is zero, then we are the child process, and must respond to the newly connected client.



NOTE Note that by using `fork`, we can in general create multi-threaded programs, not just servers. For instance, we might want to have enemy artificial intelligence routines running in a separate process parallel to the main game code. Synchronizing separate concurrent processes, however, is always the problem with multi-threading.

The section of the server that carries on a conversation with the client is similar to the corresponding section in the client. We simply use the `send` and `recv` functions to send and receive data to and from the client. Whatever the server sends with `send` is received by the client with `recv`; whatever the client sends with `send` is received by the server with `recv`.

The sample server ends the connection whenever it detects that the client has sent a string starting with the characters "quit".

Listing 8-7: `server.cc`

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>

main() {
  int mysocket;
```

```
  struct sockaddr_in addr;

  addr.sin_family = AF_INET;
  addr.sin_port = htons(5559);
  inet_aton("127.0.0.1", &addr.sin_addr);
  mysocket = socket(PF_INET, SOCK_STREAM, getprotobyname("IP")->p_proto);
  bind(mysocket, (sockaddr *)&addr, sizeof(addr));
  listen(mysocket, 10);

  struct sockaddr_in newaddr;
  while(1) {
    int newsock;
    socklen_t addrlen;
    newsock = accept(mysocket, (sockaddr *)&newaddr, &addrlen);

    if(fork()==0) {
      char msg[512];
      sprintf(msg,"Welcome to SimonServer v0.0");
      send(newsock, msg, strlen(msg), 0);

      while(1) {
        char buf[256];
        int size;
        size = recv(newsock, buf, sizeof(buf), 0);
        buf[size] = 0;
        char buf2[512];
        sprintf(buf2,"Simon says: %s", buf);
        send(newsock, buf2, strlen(buf2), 0);
        if(strncmp(buf,"quit",4)==0) {
          send(newsock, "bye", 4, 0);
          shutdown(newsock,2);
          exit(0);
        }
      }
    }
  }
  close(mysocket);
}
```

Running the Sample Server and Client

To run the client/server network program, first compile both the client and the server with **make**. Then, type **server&** and press **Enter**. The ampersand causes the server program to run in the background. Next, type **client** and press **Enter**. You should see a brief welcome message from the server. Then, anything you type will be sent to the server, and the server will repeat what you typed to demonstrate that it received the data correctly. Try starting several instances of the client program in different windows. Notice how several clients can connect simultaneously. When you are done, you should kill the server. Determine its process number with the command **ps x**, and then type **kill PID**, where PID is the process number of the server process.

Non-Blocking Operations

The socket operations as described above are blocking. That is, `accept` waits until a new connection is present before proceeding, `send` waits for the data to be completely transmitted before

proceeding, and `recv` waits for a specified amount of data to be received before continuing. This mode of operation mimics that of files.

However, with network traffic, it is sometimes more convenient to have non-blocking operations, where we can check for the presence of data, but still continue processing even if no data is present. Doing such non-blocking I/O requires you to set the `O_NONBLOCK` flag on the socket using the `ioctl` function; you then can use `select` to check for the presence (or absence) of data on the socket. See the online manual page for `socket` for more details. An alternative to polling is to have the operating system call a function of yours whenever data becomes available; to do this, you must install a signal handler for signal `SIGIO`, a topic which is also documented in the online manual.

What Data to Send

Before we close our discussion on IP networking, it's worthwhile to mention exactly what data we might wish to send over the network. In the case of interactive 3D applications, it is interesting to allow multi-user environments where the actions of one user affect the appearance or state of the virtual world for all other users. There are a number of ways we can achieve this effect. We can have a peer-to-peer network, where every connected computer can inform other computers of events which it generated, and must also receive events from all other computers to stay consistent. We could also have one master computer and several slaves; the slaves all send their local changes to the master, which then combines all the changes to form a new complete state of the world, which is then uniformly transmitted to all slaves.

The information that is transmitted will typically involve the positions, orientations, and states of all moveable or changeable objects in the world. A common misconception is that just the (x,y,z) coordinates are sufficient to describe the complete state of each object; this ignores the orientation of the object (is the bad guy facing me or not?) and the state of the object (is he wounded or at full health? Crouching, running, or jumping? With or without the plasma rifle?). Any aspect of any object that can change should be consistent across all participants' machines, and must be able to be transmitted over the network.

Another question is how often to transmit information among the computers in the network. The finest level of control and accuracy is achieved when all computers are synchronized with one another for every frame. This is probably the easiest kind of synchronization to implement. This can be quite slow, however, and faster computers spend all of their time waiting on the slower ones in this scheme. Another option is to only synchronize every few frames, but then the danger grows that the worlds will slowly drift out of sync, requiring larger synchronization phases (i.e., more transmission of data) at certain stages to ensure that everyone's view of the world is identical.

Collision Detection

The topic of *collision detection* is a non-graphical technique which nonetheless is very important for 3D graphics applications. Collision detection refers to the determination of whether two objects in the 3D virtual world overlap, or not. Collision detection generally also requires some

sort of *collision response* if we detect a collision, such as moving the objects apart again. Collision detection algorithms are important for 3D graphics applications for two reasons:

- They allow us to impart a sense of physical reality to our virtual 3D worlds, greatly enhancing the user's sense of place in the virtual world.
- They illustrate useful geometric intersection techniques which are often also needed for visual 3D graphics algorithms, such as ray tracing or shadow computation.

In this section, we'll look at collision detection in general, and then develop a very simple collision detection and response system to prevent us from walking through walls in our virtual world. We'll also populate our virtual world with other moving objects that wander throughout the world, using collision detection to keep them from walking through walls and each other.

Intersection Testing and Bounding Volumes

Collision detection essentially requires us to determine if two objects spatially overlap. Spatial overlap is the same as intersection between any parts of the two objects. Since our objects are arbitrary, non-convex polygonal solids, one way of checking would be to test every polygon of one object against every polygon of the other object, checking for intersection. With a large number of objects, this sort of exhaustive testing quickly becomes impractical because of the explosive number of tests that need to be done. Furthermore, such precise collision detection is often not even needed. It depends on the purpose of the collision detection. Often, in interactive 3D applications, a primary goal of collision detection is simply to prevent objects as a whole from moving through one another. This means that we can enclose the objects within simple, convex bounding volumes, such as spheres or boxes, and check these bounding volumes for intersection. If the bounding volumes do not intersect, then the objects do not intersect. If the bounding volumes intersect, the objects might intersect. Depending on the application, we can then perform finer polygon-to-polygon intersection tests to determine the exact collision points, or we can treat a bounding volume collision as an actual collision.

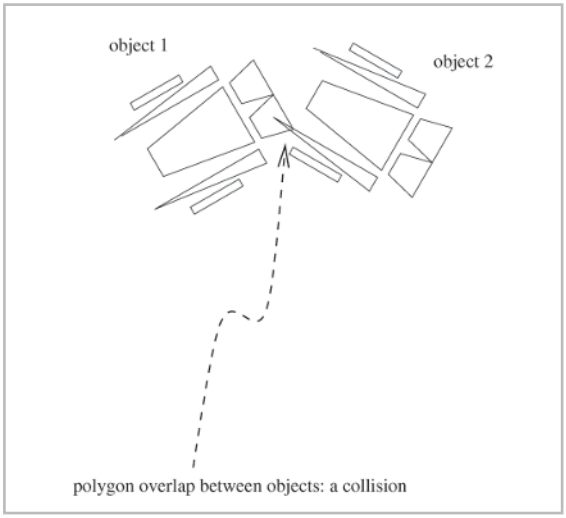


Figure 8-1: Checking for collisions can be done by checking every polygon of one object against every polygon of the other object, but this is slow.

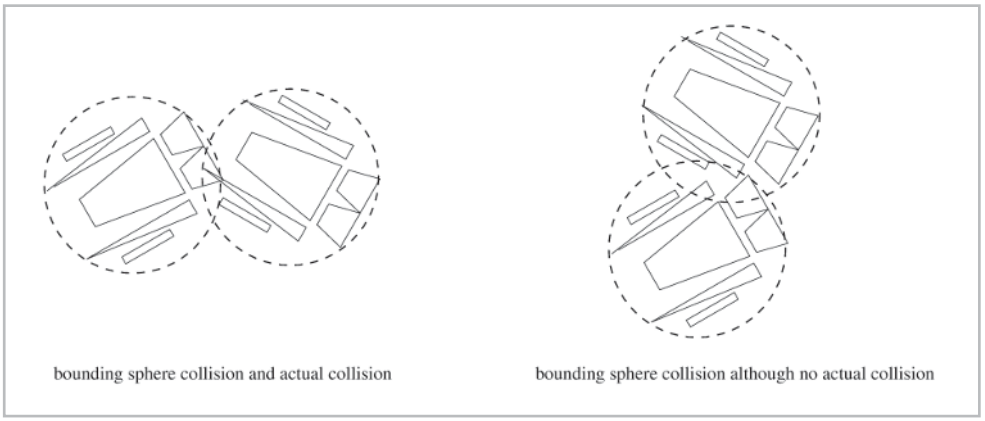


Figure 8-2: Enclosing the object in a bounding volume allows us to first check for overlap of the bounding volumes, then possibly perform a finer polygon-to-polygon check if desired.

Possible bounding volumes include spheres, axially aligned bounding boxes, or arbitrarily oriented bounding boxes. Here, we'll cover sphere-to-sphere, ray-to-polygon, ray-to-sphere, and sphere- to-polygon collisions, all of which illustrate the basic intersection techniques needed for the other methods.

Sphere-to-Sphere

Sphere-to-sphere intersection testing is almost trivial. We simply compute the distance between the centers of the two spheres. If the distance is less than the sum of the two radii of the spheres, then the spheres intersect; otherwise, they do not.

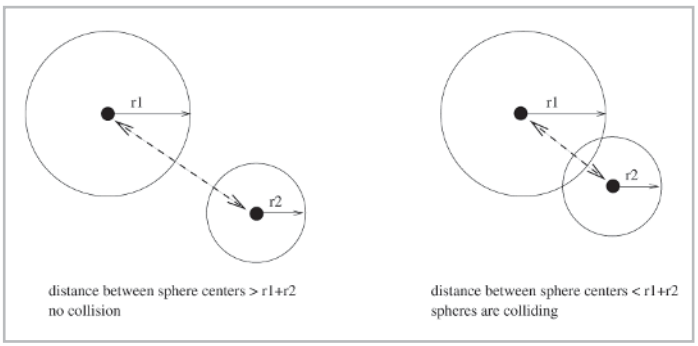


Figure 8-3: Sphere-to-sphere intersection testing is easy: use the distance between the centers of the spheres.

Ray-to-Polygon

Ray-to-polygon intersection testing is the next intersection test of interest, and is also used by another test, the sphere-to-polygon test. A ray is a directed half-line starting at one point and moving infinitely in the direction of, through, and past a second point. With ray-to-polygon collision detection, the ray goes from the old position of the object to the new position of the object. We want to determine if this ray or line of movement intersects a particular polygon. This consists of

two steps: a test to see if the ray crosses the plane of the polygon, and if so, a second test to see if the point of intersection actually lies within the polygon. For simplicity, we assume that the polygons being tested are convex.

The first test is easy. We simply evaluate the plane equation twice, once with the first point and once with the second point. If the signs of the results differ, the point moved from one side of the plane to the other. This is because evaluating a plane equation with a point's coordinates tells us the position of the point relative to the plane.

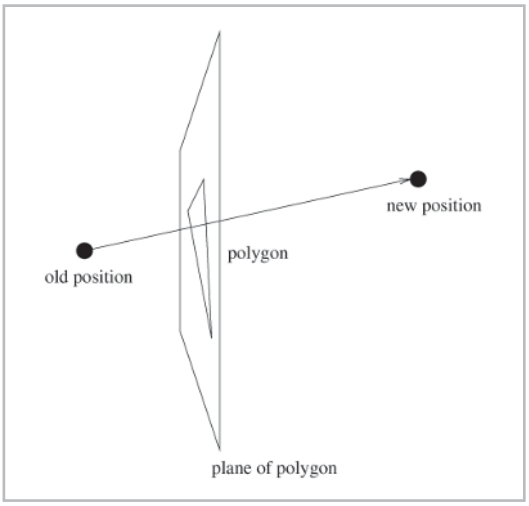


Figure 8-4: The point has crossed from one side of the polygon's plane to the other.

If the sign changes, the segment of the ray between the starting and ending points crosses the plane of the polygon, but not necessarily the polygon itself. The next test, therefore, is to compute the actual intersection point between the ray and the polygon, and see if it lies within the polygon.

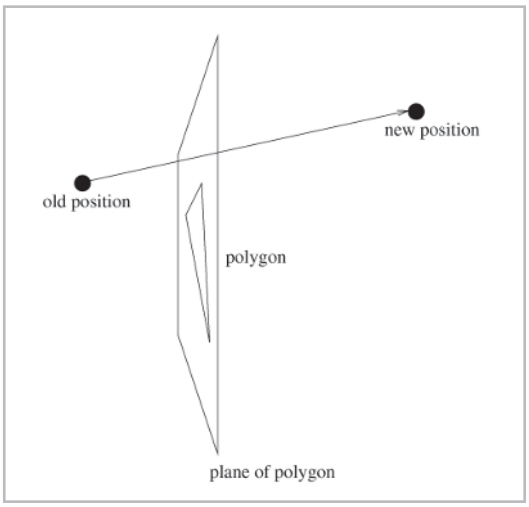


Figure 8-5: Even if the point crosses the plane, it may or may not cross the polygon. Here, the point's path does not cross the polygon.

We already know (or should know) how to compute the intersection of a line segment and a plane, as we have already needed and used this for clipping polygons to arbitrary planes. In the case of polygon clipping, each edge of a polygon is a line segment that has to be clipped against an infinite plane; this requires us to find the exact intersection of the line segment and the plane. So, we won't repeat the math here.

Given the intersection point between the ray and the plane, we must now answer the new question: does the point lie within the boundary of the polygon? The key to understanding how to solve this problem is to realize that the computation is now essentially 2D. We know an intersection point on a plane; we also know the vertices of a polygon on that plane. The easiest way to do the point-in-polygon test is to simply drop one of the coordinates x, y, or z for all points under consideration. This essentially orthogonally projects the polygon and the point onto one of the planes xy, yz, or xz. We choose which coordinate to drop based on the plane's normal vector; the largest component gets dropped, because a large normal vector component means there is little variation or tilt of plane in that direction. Dropping the largest normal vector coordinate in this manner yields the largest possible projected 2D polygon.

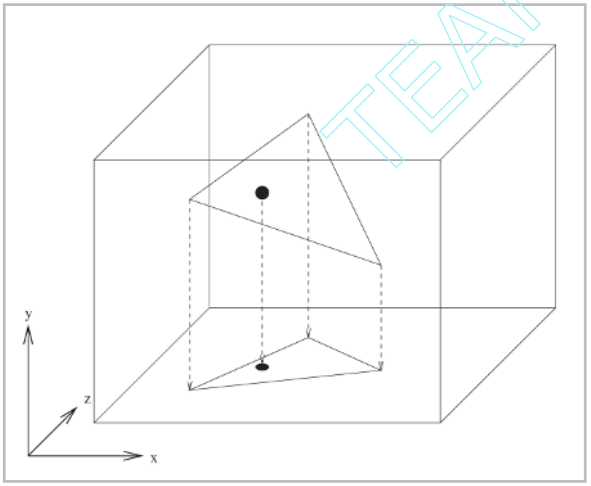


Figure 8-6: Projecting the 3D polygon and the 3D plane intersection point into 2D to test for containment in 2D. Here, we orthogonally project the polygon onto the xz plane, but we can also project onto the xy or the yz plane; we use the polygon normal vector to choose which plane to use.

After projecting the polygon into 2D in this manner, we can then use the `side_of_point` function (used during analytic 2D polygon clipping) to determine if the point is on the inside side of each of the polygon's edges. If so, then the point lies in the polygon in 2D, and therefore also lies in the polygon in 3D, indicating that the ray under question does indeed intersect the 3D polygon. If the point does not lie in the polygon in 2D, then the point does not lie in the polygon in 3D, and no collision between the ray and the polygon occurred.

Two small but tricky implementation details arise with this method: the clockwise/counterclockwise orientation of the 2D polygons we create in this manner, and the y axis orientation. Depending on the world space orientation of the polygon's normal vector, the projected 2D polygon may either be clockwise or counterclockwise when orthogonally projected onto 2D. Also, dropping one coordinate (x, y, or z) means that we use the remaining two coordinate axes as a new 2D coordinate system, but these 2D coordinate systems have the positive y axis going upward,

TEAM-FLY

whereas the `side_of_point` function developed for screen-space polygon clipping assumes a reversed-*y* coordinate system with positive values of *y* going downward. The class `l3d_polygon_3d_collidable` contains commented code for dealing exactly with these details.

Using each edge of a polygon as the ray in the ray-to-polygon test allows us to test for polygon-to-polygon penetration. If any one edge of a polygon intersects the other polygon, the two polygons intersect; otherwise, they do not. Also, ray-to-polygon testing can be used for shadow generation with light maps; the ray goes from the light source to each lumel in the scene; we check this light ray against every polygon in the scene to see if the light reaches the lumel or not. See Chapter 2 for a discussion of light mapped shadows.

Ray-to-Sphere

Ray-to-sphere intersection testing tests whether a ray intersects a sphere. We're going to look at a special case of this, which simply checks to see if a particular line segment (part of a ray) intersects the sphere. One way to do this is as follows. Call the center of the sphere *c*, and the endpoints of the line segment being tested *a* and *b*. Then, compute the vector *c–a*, which is the vector from the start point of the segment to the sphere center. Next, compute the normalized vector *v=b–a*, which is the unit vector from *a* to *b*. Take the dot product of these two vectors. Recall that the dot product of two vectors, where one is of unit length, gives the length of the projection of the non-unit vector onto the unit vector. Call the dot product of these two vectors *t*. *t* then represents the distance we must travel from *a* along the vector *b–a* in order to reach the point on the line which is perpendicularly nearest to the sphere's center. This is because the dot product is a perpendicular vector projection; here, we are perpendicularly projecting the vector to the sphere onto the line segment, thus giving us the distance along the segment to this perpendicular point.

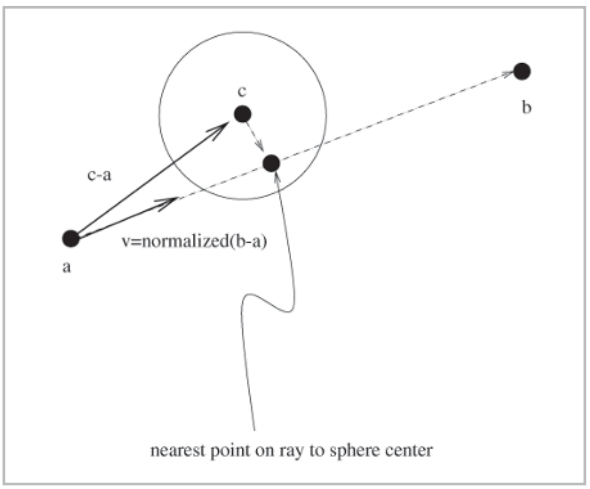


Figure 8-7: Ray-to-sphere testing uses a vector projection to find the nearest point on the ray to the sphere, then checks the distance of this nearest point to the sphere's center.

If the value of *t* is less than 0, then the nearest point to the sphere along the infinite line lies before point *a*; thus, point *a* is the nearest point still on the line segment. If *t* is greater than 1, then the nearest point on the infinite line lies after point *b*, meaning that the nearest point on the seg-

ment is point b. Otherwise, the nearest point lies between a and b, so we scale v by t and add it to point a to get the intersection point $a+vt$.

At this point, we have the point along the line segment that is located nearest to the sphere's center. We then compute the distance between this nearest point and the sphere's center. If this distance is less than the sphere's radius, then the point lies within the sphere, indicating that the line segment intersects the sphere. If the distance is greater than the sphere's radius, then the nearest point lies outside of the sphere, meaning that the entire line segment lies outside of the sphere, and does not intersect it.

Sphere-to-Polygon

Sphere-to-polygon intersection testing is based on ray-to-polygon and ray-to-sphere intersection tests. We can distinguish between two kinds of collision: moving collisions and static collisions. We say that a moving collision occurs if the center of the sphere moves from one side of the polygon to the other side, and the intersection of the ray going from the old sphere center to the new sphere center lies within the polygon. In other words, a moving collision is detected in the same way as a ray-to-polygon intersection, using the old and new positions of the sphere's center as the endpoints of the ray. Therefore, we already know how to perform this kind of collision test.

We say that a static collision occurs if the sphere's center does not cross the plane of the polygon, but the sphere nevertheless moves close enough to the polygon that it overlaps. This requires some new computation: we must determine if any part of the polygon lies within the sphere's radius. If so, a collision has occurred between the sphere and the polygon; otherwise, none has occurred.

The term "static collision" comes from the fact that a sphere might be embedded in a polygon for several frames; the collision between the plane and the polygon did not necessarily just occur, but might have been and might continue to stay that way for a while. Thus, the collision is a static condition that currently exists. (An example of a continuous static collision is if the sphere is resting halfway within a portal polygon.) The term "moving collision" comes from the fact that a sphere moving from one side of a polygon completely through to the other side has caused a collision because of the sphere's movement between the last frame and the current one. This sort of collision occurs between one frame and the next; unlike a static collision, it cannot exist for several frames.

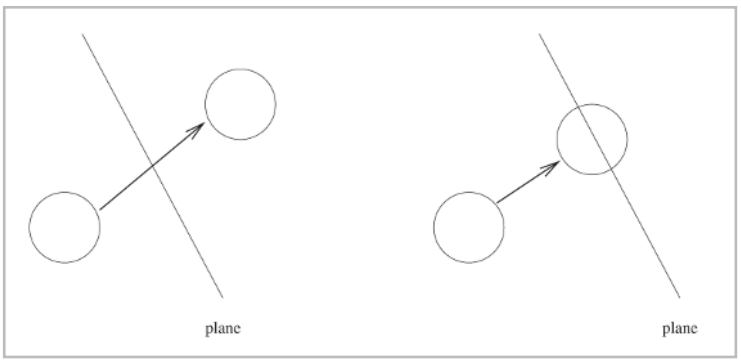


Figure 8-8: A moving collision (left) and a static collision (right).

Detecting if a static collision occurred can be done in two steps. First of all, we compute where the sphere intersects the plane of the polygon. If this intersection point lies within the polygon, the sphere has collided with some interior portion of the polygon. Secondly, if the first test fails, we must also compute if any edge of the polygon lies within the sphere. If so, then the sphere has collided with some edge of the polygon. If both tests fail, the sphere and the polygon do not intersect.

The first test computes the intersection between the sphere and the plane of the polygon. We do this by computing the distance between the polygon and the plane, by evaluating the plane equation with the center point of the sphere. Remember, the plane normal must be normalized for this to work. If this computed distance is greater than the sphere's radius, then the sphere does not intersect the polygon's plane at all, so no collision can occur, and we are done. Otherwise, if the distance is less than the sphere's radius, we calculate the point of intersection by starting at the sphere's center and adding a displacement vector with a direction exactly opposite to the plane's normal vector, and with a magnitude equal to the computed distance. This computation moves from the sphere's center exactly along the plane's reversed normal vector towards the plane, ending up at an intersection point exactly on the plane. This location is the first point of contact where the sphere would intersect the plane of the polygon.

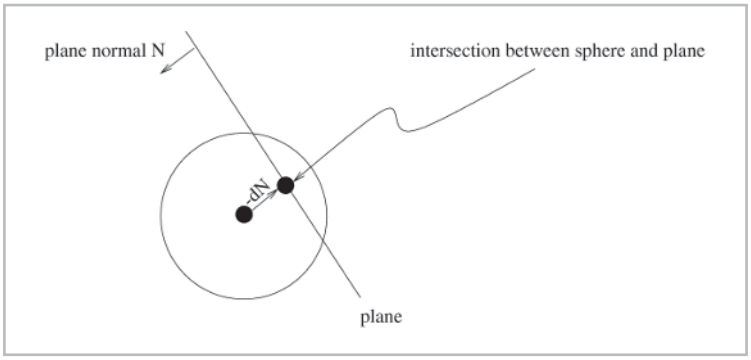


Figure 8-9: Computing the intersection point between a sphere and a plane. We start at the sphere's center, and travel along the negative plane normal vector by a distance equal to the sphere's distance from the plane.

We then test this sphere intersection point for containment within the polygon, using the same orthogonal 2D projection technique from the ray-to-polygon test. If the point lies within the polygon, a collision occurred; if not, no collision occurred with the sphere and the interior of the polygon. But the sphere might still be colliding with the edge of the polygon, leading to the last test.

The second and last test computes to see if any edge of the polygon lies within the sphere, relying on the ray-to-sphere test we looked at earlier. To do this, we process each edge of the polygon individually. For each edge, we compute the nearest point on each edge to the sphere's center, using the method we used for ray-to-sphere intersection. After finding the nearest point on each edge to the sphere's center, we take the nearest point of all edges, which is then the nearest point to the sphere on the entire polygon border. If this nearest point lies within the sphere, then some part of some polygon edge lies within the sphere, and a collision with the polygon border has occurred. If this nearest point lies outside of the sphere, then all points on all polygon edges lie outside of the

sphere, and no collision with the polygon boundary has occurred. Since the first test returned that no collision occurred with the polygon interior, we know that no collision occurred at all between the sphere and the polygon.

Tunneling and Sweep Tests

We implicitly addressed the problem of *tunneling* with the last collision test, sphere-to-polygon, but it is important enough that we should explicitly mention it. The term "tunneling" refers to the problem that objects moving with a high speed in a discrete computer simulation move from one location to a spatially distant location instantaneously, without ever occupying any of the space in between the old and the new positions. When the distance traveled by an object per frame remains small relative to the size of the object and other objects, then no problems arise. But as soon as objects start covering larger distances per time frame, the danger exists that the large discrete steps taken by the object might cause it to move straight past objects with which it should have collided.

A simple example of this phenomenon is a fast-moving sphere approaching a plane. As long as the sphere moves in discrete steps that are smaller than its diameter, we have no problem. But if the sphere hops by a distance greater than its diameter, it can hop from one side of the plane to the other without ever actually intersecting the plane.

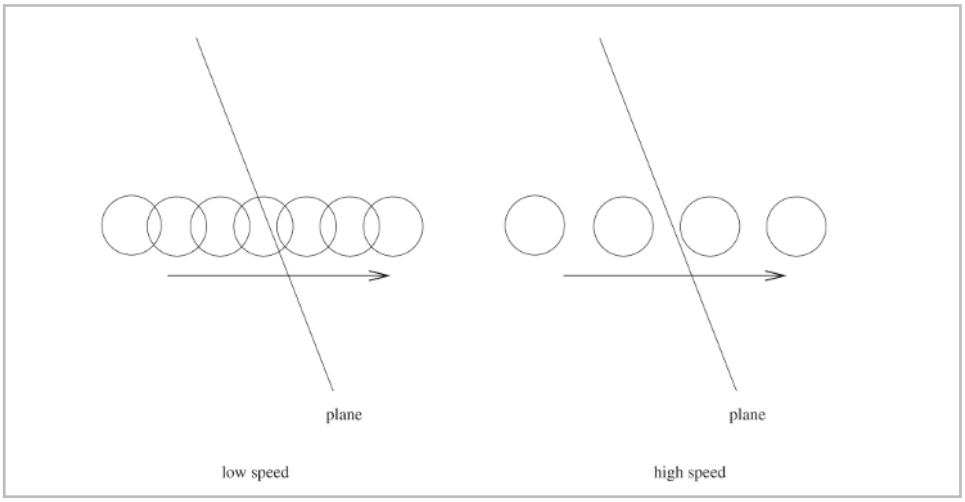


Figure 8-10: At low speeds, collisions are detected (left), but a high-speed sphere moves from one side of a wall to the other side without ever touching it (right).



NOTE The term tunneling comes from the field of quantum mechanics, in which it has been experimentally established that electrons tunnel through barriers that are otherwise impenetrable [MOEL99].

We implicitly solved this problem earlier by introducing the concept of moving collisions and static collisions for the sphere-to-plane test. A moving collision detects exactly the case of a sphere hopping across a plane.

The problem also exists with sphere-to-sphere collisions. Two fast-moving spheres might cross paths without ever colliding with one another, although they should actually have collided at an earlier point in time.

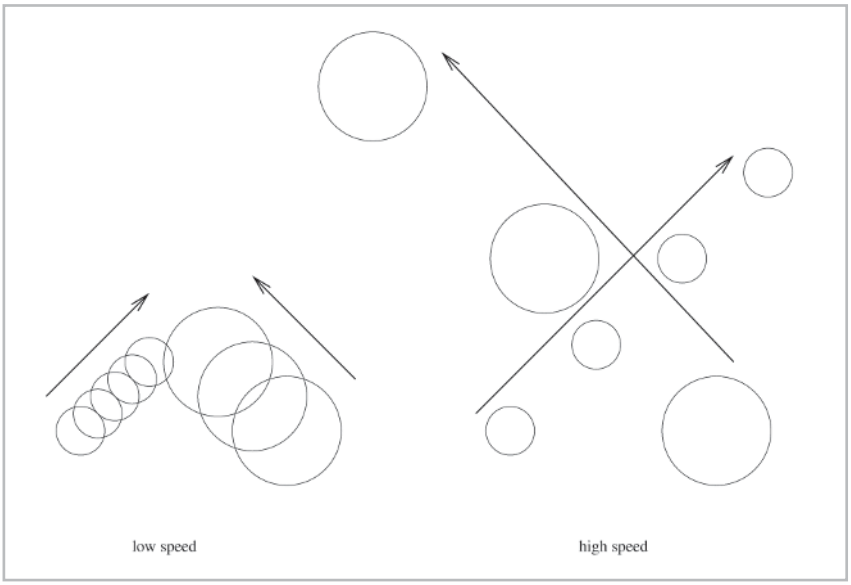


Figure 8-11: Two high-speed spheres sailing past one another without registering a collision.

For the sphere-to-sphere case, and in general for other bounding-volumes-to-bounding-volume tests as well, we can detect this situation by using a *sweep test*. A *sweep* refers to the volume of space covered by a moving object between the object's old position and its new position. Sweep tests therefore must detect if a collision occurred at any point along the object's trajectories. Let's derive a sweep test for sphere-to-sphere intersection, which illustrates the general methods used for developing such tests.

Assume we have two moving spheres P and Q. Sphere P was initially located at location P_0, and travels at a velocity V_p towards its new position P_1. Similarly, sphere Q begins at Q_0 and travels with velocity V_q towards its new position Q_1. Let us adopt the convention that at time t=0, both spheres were at their starting points, and at time t=1, both spheres were at their new positions. This is standard parameterization. Then, the question is, is there any t such that the spheres collided?

We define a sphere collision in terms of the distance between the spheres' centers. We can compute the distance between two points as the square root of the dot product of an offset vector with itself, where the offset vector is the vector from one sphere's center to the other. Thus, at a given time t, the distance between P and Q is given by:

Equation 8-1

$$\begin{aligned} d(t) &= \sqrt{o(t) \cdot o(t)} \\ o(t) &= (Q_0 + tV_q) - (P_0 + tV_p) \end{aligned}$$

Here, d is the distance, and o is the offset vector. Notice that o depends on time, since the spheres are moving. We take the current position of Q and subtract the current position of P to get the offset vector for a particular time, which we then use to compute the distance.

It turns out that the computations are easier if we take the square distance instead of the distance:

Equation 8-2
$$d^2(t) = o(t) \cdot o(t)$$

Furthermore, we can rewrite o in vector form as follows:

Equation 8-3
$$\begin{aligned} o(t) &= t(V_q - V_p) + (Q_0 - P_0) \\ &= (t\overrightarrow{V_{pq}} + \overrightarrow{P_0Q_0}) \end{aligned}$$

The vector V_{pq} is just a shorthand way of writing V_q–V_p. The vector P_0Q_0 is shorthand for Q_0–P_0. Remember, these are all known quantities.

By taking the dot product of the time-dependent offset vector o with itself, we can know the squared distance between the two sphere centers for any value of t. To detect a collision, we want to see if there exists any value of t where this squared distance equals the squared sum of the sphere's radii. If there is such a t, then at this time the spheres would be touching. Mathematically, we can express this collision condition as follows.

Equation 8-4
$$o(t) \cdot o(t) = (r_p + r_q)^2$$

We can rearrange the terms slightly:

Equation 8-5
$$o(t) \cdot o(t) - (r_p + r_q)^2 = 0$$

Then, expanding the terms for o:

Equation 8-6
$$t^2(\overrightarrow{V_{pq}} \cdot \overrightarrow{V_{pq}}) + t(2\overrightarrow{P_0Q_0} \cdot \overrightarrow{P_0Q_0}) + (\overrightarrow{P_0Q_0} \cdot \overrightarrow{P_0Q_0} - (r_p + r_q)^2) = 0$$

All values are known except for t. This is a quadratic equation in t, so to solve for t, we can use the quadratic formula, which you may recall from high school algebra:

Equation 8-7
$$-b \pm \sqrt{\frac{b^2 - 4ac}{2a}}$$

In the quadratic formula, a is the coefficient of the quadratic term t^2, b is the coefficient of the linear term t, and c is the constant term. The portion under the radical, (b^2–$4ac$)/$2a$, is called the *discriminant*, and determines the number of solutions to the quadratic equation. If it is negative, there are no solutions to the equation, meaning that the spheres do not ever collide if they travel along their current paths. If it is zero, then there is exactly one solution, and if it is positive, then

there are two solutions. (Recall that when a quadratic equation is plotted it yields a parabola; finding the roots of the equation is equivalent to determining when the parabola crosses the x axis, which can be either zero, one, or two times.)

So, if the discriminant is non-negative, we plug in the coefficients from Equation 8-6 and solve for t. If t lies between 0 and 1, the spheres do indeed collide at time t. If there are two solutions for t, then we take the smaller value, because the smaller value represents when the spheres contact when they are moving toward each other, and the greater value represents when the spheres are just leaving each other.

If t does not lie between 0 and 1, then the spheres would eventually collide, but they do not collide along the small segment of their paths traveled during the time t=0 to t=1, which is all that interests us for this frame. So in this case, no collision occurred for this frame.

Multiple Simultaneous Collisions and Collision Response

This section deals with the two topics together: multiple simultaneous collisions and collision response. We treat these topics together because they influence one another to some degree. Once we detect that a collision occurred, we should do something about it. In general, this is completely application-specific. For instance, if we are modeling a game where the player runs around collecting coins to earn points, then when the player collides with a coin, we probably simply cause the coin to disappear and add it to the player's score. Collision response in this case is very simple.

Although collision response is application-specific, there is a very common class of collision response that has a very simple goal: prevent objects from penetrating one another, in a physically realistic way. This prevents objects from moving through one another, thus modeling the ordinary physical reality that no two objects can occupy the same space at the same time.

Maintaining some semblance of physical reality when preventing collisions is important to make the simulation believable. For instance, a physically unrealistic but very simple way of preventing objects from colliding would be to instantaneously stop the motion of any object if that motion would cause a collision. While this accomplishes the goal of preventing object penetrations, ordinary objects change their positions, velocities, and accelerations when they collide; they don't just stop all motion instantaneously.

Allowing Penetration

One way of responding to collisions is to detect and allow object penetration, and to model the result of the penetration as a spring force [SHAN98]. The penetrated object therefore pushes back on the penetrating object like a spring, responding with more force with deeper penetration. With such a system, we could account for multiple simultaneous collisions by adding the contributions of each spring force.

A problem with this method is that if our time step is too large, an object might be allowed to penetrate an object to a great depth, which would produce a tremendous spring force and cause the object to fly back at a great speed. The best solution is to run the simulation code as fast as possible so that the time steps are sufficiently small enough to prevent objects from moving so much in one

time step that they penetrate deeply. But the problem can still crop up if the code slows down, forcing the time step to increase. We can then compensate for the effects by using damping forces after collision to ensure that the new velocities remain within reasonable bounds [SHAN98].

Avoiding Penetration with Temporal Search

Another method of collision response is to forbid object penetration with temporal search, and to resolve each collision under the assumption that the colliding bodies are rigid. This usually implies the use of rigid body dynamics; see the section later in this chapter titled "Physics."

Key to this method is the prevention of collisions. To do this, if we detect a collision, then we back up the clock and determine the exact time at which the collision occurred. We can do this analytically by solving for the geometric intersection time, as we did with the sweep test earlier. Alternatively, if we do not know how to analytically solve for the collision time, we can perform a binary search through time to find the collision time. This assumes that we know that the objects are indeed colliding (for instance, because they are currently overlapping) but don't know exactly at what point in time they first collided. By itself, the temporal search method does not address tunneling issues, and should be combined with some sort of (possibly inexact) sweep testing.

The idea behind the binary temporal search method is as follows. We know that the objects weren't colliding in the previous frame, and they are now. We also know the clock value at the previous non-colliding frame, and the clock value at the current colliding frame. So, based on the previous non-colliding positions and velocities, we test the positions at different time values lying between the non-colliding time value and the colliding time value. We search for the earliest time value at which the objects were still not colliding, within a given temporal tolerance such as 0.1 seconds. A binary search is a natural solution here. If there was no collision at time t, and there was a collision at time $t+dt$, then we check at time $t+0.5dt$. If the objects are still colliding, we back up again in time by one-half of the previous temporal interval, to $t+0.25dt$; if no collision was found, we step forward in time to $t+0.75dt$. We stop searching whenever the bounds of the temporal search space differ by less than the tolerance.

Once we find the exact time (within the specified tolerance) at which a collision occurred, we then must alter the velocities of the objects to prevent them from moving into one another. This altering of the velocities should follow some physical laws to appear realistic, simulating the impact of the two bodies (see the section titled "Physics"). By altering the velocities in a physically realistic manner, the objects will bounce away from each other at the exact instant of collision, as opposed to afterwards as with the spring method. The new object velocities then cause the objects to move in new directions, which avoids the collision.

Multiple simultaneous collisions in this system do not really exist; instead, we sort all collisions temporally and resolve them on a pair-wise basis. We first find all collisions among all objects in a pair-wise manner. In other words, we make a list of pairs of objects (A,B), where an entry (A,B) in the list implies that A collides with B. We then determine the exact time at which each such collision pair occurred, and sort these by time. We handle the earliest collision first, altering the velocities of the objects to prevent the first collision. Then, we re-evaluate all other collision pairs based on the new positions of the objects A and B given their new velocities. New collisions can occur and old collisions can disappear as a result of the first collision response.

After updating the collision list, we then again sort the list temporally and resolve the collisions in temporal order until the list is empty.

It is possible that so many bodies collide in such a complex manner that the collision list cannot be resolved in a reasonable amount of time; inherent numerical inaccuracy in the computations also compounds the problem. We can try to circumvent this by preventing the routine from looping more than a fixed number of times; if we exceed this count and still weren't able to resolve all collisions, we can then resort to an emergency spatial search to position each object so that it no longer collides with any other. We can do this by progressively bouncing each object farther and farther away from the objects with which it collides, until it has bounced so far away that it no longer collides with them. The problem here is that we might have to bounce the objects very far from one another, leading to physically unrealistic super-explosive behavior with complex collisions. Again, use of a damping force after collision resolution can help reduce the velocities.

Now that we've covered some of the important collision detection principles, let's look at some sample classes and code which implement collision detection.

Class l3d_collidable

Class `l3d_collidable` represents an object which can be tested for collision against another collidable object.

Abstract method `detect_collision_with` takes another `l3d_collidable` object as a parameter. It returns whether or not the current object collides with the other object.

Member variables `old_position` and `new_position` represent the movement of the object. `old_position` is the position in the previous frame; `new position` is the new position to which the object wants to move. A collision may prevent the object from reaching its originally specified `new_position`. In this case, `new_position` is changed to reflect the actual new position to which the object is allowed to move. `new_position` may be changed several times, if the object collides with many other objects along its path.

Member variable `bounce` is a vector representing a spring-like change in velocity resulting from the collision. (Note that this modeling is oversimplified; see the section titled "Physics.")

Member variable `prevent_collision` indicates whether the object should automatically correct a collision once one is detected, so that the objects no longer collide.

Listing 8-8: `collide.h`

```
#ifndef __COLLIDE_H
#define __COLLIDE_H

#include "../tool_os/memman.h"
#include "../math/vector.h"
#include "../geom/point/point.h"

class l3d_collidable {
  public:
    l3d_point old_position, new_position;
    l3d_vector bounce;
    int prevent_collision;
```

```
    virtual int detect_collision_with(l3d_collidable *target) {return 0; };
};

#endif
```

Class l3d_collidable_sphere

Class `l3d_collidable_sphere` is a sphere that can test itself for collision against other objects.

Member variable `collision_sphere_radius` is the radius of the sphere.

Overridden method `detect_collision_with` detects the collision of the sphere with various types of other objects. Implemented are the sphere-plane, sphere-polygon, and sphere-sphere tests described earlier. If member variable `prevent_collision` is true, then the sphere's position is corrected after a collision so that it no longer collides with the other object. Also, the `bounce` vector is set to indicate the exact correction that was done.

Listing 8-9: `colsph.h`

```
#ifndef __COLSPH_H
#define __COLSPH_H

#include "collide.h"

class l3d_collidable_sphere :
      virtual public l3d_collidable
{
  public:
    l3d_real collision_sphere_radius;
    int detect_collision_with(l3d_collidable *target);
};

#endif
```

Listing 8-10: `colsph.cc`

```
#include "colsph.h"
#include "../geom/plane/plane.h"
#include "../geom/polygon/p3_coll.h"
#include "../tool_os/memman.h"

int l3d_collidable_sphere::detect_collision_with(l3d_collidable *target)
{
  l3d_plane *plane;
  l3d_polygon_3d_collidable *poly;
  l3d_collidable_sphere *sphere;

  if( (plane=dynamic_cast<l3d_plane *>(target)) ) {
    l3d_real dist;

    int status;
    status = plane->intersect_with_segment(old_position, new_position);

    if (
      plane->side_of_point(old_position) >= 0 &&
      plane->side_of_point(new_position) < 0 )
    {
```

TEAM-FLY

```
#define EPSILON_COLLISION float_to_l3d_real(0.1)

      bounce.set
      (plane->intersection.X_ +
       l3d_mulrr(plane->a, collision_sphere_radius + EPSILON_COLLISION) -
       new_position.X_,

       plane->intersection.Y_ +
       l3d_mulrr(plane->b, collision_sphere_radius + EPSILON_COLLISION) -
       new_position.Y_,

       plane->intersection.Z_ +
       l3d_mulrr(plane->c, collision_sphere_radius + EPSILON_COLLISION) -
       new_position.Z_,

       int_to_l3d_real(0));

      if(prevent_collision) {
        new_position = new_position + bounce;
      }

      return 1;
    }else if ((dist=plane->side_of_point(new_position)) <collision_sphere_radius
               && dist>0
              )
    {

      bounce.set
      (l3d_mulrr(plane->a,
                 collision_sphere_radius - dist + EPSILON_COLLISION),
       l3d_mulrr(plane->b,
                 collision_sphere_radius - dist + EPSILON_COLLISION),
       l3d_mulrr(plane->c,
                 collision_sphere_radius - dist + EPSILON_COLLISION),
       int_to_l3d_real(0));

      if(prevent_collision) {
        new_position = new_position + bounce;
      }

      return 2;

    }
    else return 0;

  }
  else if ( (poly = dynamic_cast<l3d_polygon_3d_collidable *>(target)) ) {

    l3d_real dist;

    int status;
    status = poly->plane.intersect_with_segment(old_position, new_position);

    if (
      poly->plane.side_of_point(old_position) >= 0 &&
      poly->plane.side_of_point(new_position) < 0 &&
      poly->contains_point(poly->plane.intersection) )
    {

      bounce.set
```

```
  (poly->plane.intersection.X_ +
   l3d_mulrr(poly->plane.a, collision_sphere_radius + EPSILON_COLLISION) -
   new_position.X_,

   poly->plane.intersection.Y_ +
   l3d_mulrr(poly->plane.b, collision_sphere_radius + EPSILON_COLLISION) -
   new_position.Y_,

   poly->plane.intersection.Z_ +
   l3d_mulrr(poly->plane.c, collision_sphere_radius + EPSILON_COLLISION) -
   new_position.Z_,

   int_to_l3d_real(0));

  if(prevent_collision) {
    new_position = new_position + bounce;
  }

  return 1;
}else if ((dist=poly->plane.side_of_point(new_position)) <collision_sphere_radius
          && dist>0
         )
{

  //- check for collision with polygon interior

  l3d_point plane_intersection;
  plane_intersection.set
  (new_position.X_ + l3d_mulrr(dist, -poly->plane.a),
   new_position.Y_ + l3d_mulrr(dist, -poly->plane.b),
   new_position.Z_ + l3d_mulrr(dist, -poly->plane.c),
   int_to_l3d_real(1));

  if(poly->contains_point(poly->plane.intersection)) {
    if(poly->contains_point(plane_intersection)) {

      bounce.set
      (l3d_mulrr(poly->plane.a,
                 collision_sphere_radius - dist + EPSILON_COLLISION),
       l3d_mulrr(poly->plane.b,
                 collision_sphere_radius - dist + EPSILON_COLLISION),
       l3d_mulrr(poly->plane.c,
                 collision_sphere_radius - dist + EPSILON_COLLISION),
       int_to_l3d_real(0));

      if(prevent_collision) {
        new_position = new_position + bounce;
      }

      return 2;
    }

    //- check for collision with polygon edges

    l3d_real smallest_dist = int_to_l3d_real(9999);
    l3d_point nearest_point;
    for(int ivtx=0; ivtx<poly->clip_ivertices->num_items; ivtx++) {
      l3d_point a =
        (**poly->vlist)[(*poly->clip_ivertices)[ivtx].ivertex].
        transformed;
```

```
        int next_ivtx = poly->next_clipidx_right
                        (ivtx, poly->clip_ivertices->num_items);
        l3d_point b =
          (**poly->vlist)[(*poly->clip_ivertices)[next_ivtx].ivertex].
          transformed;

        l3d_point p = new_position;
        l3d_vector c = p - a;
        l3d_vector V = normalized(b-a);
        l3d_real d = l3d_sqrt(dot(b-a,b-a));
        l3d_real t = dot(V,c);

        l3d_point pt;

        if(t<0) {
          pt = a;
        }else if(t>d) {
          pt = b;
        }else {
          pt = a + V*t;
        }

        l3d_real temp_dist;
        temp_dist = l3d_sqrt(dot(pt-new_position,
                                 pt-new_position));
        if(temp_dist <smallest_dist)
        {
          smallest_dist = temp_dist;
          nearest_point = pt;
        }
      }

      if( smallest_dist < collision_sphere_radius) {
        bounce.set
        (l3d_mulrr(poly->plane.a,
                   collision_sphere_radius - dist + EPSILON_COLLISION),
         l3d_mulrr(poly->plane.b,
                   collision_sphere_radius - dist + EPSILON_COLLISION),
         l3d_mulrr(poly->plane.c,
                   collision_sphere_radius - dist + EPSILON_COLLISION),
         int_to_l3d_real(0));

        if(prevent_collision) {
          new_position = new_position + bounce;
        }

        return 2;

      }

      else return 0;
    }
    else return 0;
  }
  return 0;
}
else if ( (sphere = dynamic_cast<l3d_collidable_sphere *>(target)) ) {

  int result = 0;
```

```
    l3d_real dist;
    dist = l3d_sqrt(dot(new_position - target->new_position,
                        new_position - target->new_position));

    if(dist < collision_sphere_radius + sphere->collision_sphere_radius)
    {
      result = 1;

      l3d_real bounce_dist = collision_sphere_radius
                           + sphere->collision_sphere_radius
                           - dist;

      bounce = normalized(new_position - sphere->new_position);
      bounce.set(l3d_mulrr(bounce.X_, bounce_dist + EPSILON_COLLISION),
                 l3d_mulrr(bounce.Y_, bounce_dist + EPSILON_COLLISION),
                 l3d_mulrr(bounce.Z_, bounce_dist + EPSILON_COLLISION),
                 int_to_l3d_real(0));

      if(prevent_collision) {
        new_position = new_position + bounce;
      }
    }

    return result;
  }else {
    printf("unknown collidable test sphere with ???");
    return 0;
  }

}
```

Class l3d_polygon_3d_collidable

Class `l3d_polygon_3d_collidable` is a polygon which can be tested for collision with other objects.

Method `contains_point` determines if a particular point, which must lie within the plane of the polygon, lies within the polygon itself. It uses the orthogonal projection method, which we developed for the ray-polygon test, to eliminate one of the dimensions from consideration, then to perform the point-in-polygon test in 2D. Method `contains_point` is called during the sphere-polygon intersection test.

Listing 8-11: `p3_coll.h`

```
#ifndef __P3_COLL_H
#define __P3_COLL_H
#include "../../tool_os/memman.h"

#define __ACTIVE__P3_COLL_H

#include "p3_clip.h"
#include "../plane/plane.h"
#include "../../dynamics/collide.h"

class l3d_polygon_3d_collidable :
      virtual public l3d_polygon_3d_clippable,
      virtual public l3d_collidable
{
  public:
```

```
  l3d_polygon_3d_collidable(void) : l3d_polygon_3d_clippable() {};

    l3d_polygon_3d_collidable(int num_pts) :
    l3d_polygon_3d(num_pts),
    l3d_polygon_2d(num_pts),
    l3d_polygon_3d_clippable(num_pts)
  {};

    int contains_point(const l3d_point &point);

    l3d_polygon_3d_collidable(const l3d_polygon_3d_collidable &r);
    l3d_polygon_2d *clone(void);
};

#undef __ACTIVE__P3_COLL_H

#endif
```

Listing 8-12: p3_coll.cc

```
#include "p3_coll.h"
#include "../../tool_os/memman.h"

int l3d_polygon_3d_collidable::contains_point(const l3d_point& point) {
  int collindex[2];
  int inside_side;

  if(l3d_abs(plane.a) > l3d_abs(plane.b) &&
      l3d_abs(plane.a) > l3d_abs(plane.c))
  {
    //- smallest variation in the a (x) direction (largest normal component);
    //- drop it
    collindex[0] = 2; //- z axis becomes new 2D x axis
    collindex[1] = 1; //- y axis becomes new 2D y axis
    //- NOTE: look at LHS system. drop x. you must use "y" as the vertical
    //- axis and "z" as horiz. in order for left-right orientation to stay
    //- the same.

    //- if the dir. of the normal vector component you dropped is positive,
    //- the orientation of the orthogonally-projected poly is clockwise
    //- (the normal case for a LHS coordinate system) else counterclockwise.
    //- BUT!! side_of_point assumes SCREEN-SPACE coords, i.e. +ve y axis is
    //- DOWN, but here we are working in 3D space with one axis dropped,
    //- giving an orthographically projected point with +ve y axis going UP.
    //- so we flip our interpretation of "inside" and "outside" from the
    //- result returned by side_of_point: we treat -1 as inside for normal
    //- clockwise polys, and treat +1 as inside for reversed counterclockwise
    //- polys.

    if(plane.a > 0) inside_side = -1; else inside_side = 1;
  }else if(l3d_abs(plane.b) > l3d_abs(plane.a) &&
           l3d_abs(plane.b) > l3d_abs(plane.c) )
  {
    //- smallest variation in the b (y) direction (largest normal component);
    //- drop it
    collindex[0] = 0; //- x axis becomes new 2D x axis
    collindex[1] = 2; //- z axis becomes new 2D y axis

    //- if the dir. of the normal vector component you dropped is positive,
    //- the orientation of the orthogonally-projected poly is clockwise
    //- (the normal case for a LHS coordinate system) else counterclockwise.
```

```
  //- BUT!! side_of_point assumes SCREEN-SPACE coords, i.e. +ve y axis is
  //- DOWN, but here we are working in 3D space with one axis dropped,
  //- giving an orthographically projected point with +ve y axis going UP.
  //- so we flip our interpretation of "inside" and "outside" from the
  //- result returned by side_of_point: we treat -1 as inside for normal
  //- clockwise polys, and treat +1 as inside for reversed counterclockwise
  //- polys.
  if(plane.b > 0) inside_side = -1; else inside_side = 1;

}else {
  //- smallest variation in the c (z) direction (largest normal component);
  //- drop it

  collindex[0] = 0; //- x axis becomes new 2D x axis
  collindex[1] = 1; //- y axis becomes new 2D y axis

  //- here, because the +ve z-axis is pointing into the screen, then
  //- if plane.c > 0 then the polygon is pointing AWAY from the
  //- viewer, meaning that the orientation of the polygon is reversed
  //- (i.e. clockwise), so normally inside_side would be -1. BUT,
  //- due to the reversed-y system expected by side_of_point(),
  //- inside_side gets negated again, giving 1.
  if(plane.c > 0) inside_side = 1; else inside_side = -1;
}

int idx0, idx1;
idx0 = 0;
idx1 = next_clipidx_right(idx0, clip_ivertices->num_items);
l3d_point ortho_point, ortho_p1, ortho_p2;

ortho_point.set(point.a[collindex[0]],
                point.a[collindex[1]],
                int_to_l3d_real(0),
                int_to_l3d_real(1));

int status=1;

int done=0;
while(!done) {
  ortho_p1.set
  ( (**vlist)[(*clip_ivertices)[idx0].ivertex].transformed.a[collindex[0]],
    (**vlist)[(*clip_ivertices)[idx0].ivertex].transformed.a[collindex[1]],
    int_to_l3d_real(0),
    int_to_l3d_real(1) );
  ortho_p2.set
  ( (**vlist)[(*clip_ivertices)[idx1].ivertex].transformed.a[collindex[0]],
    (**vlist)[(*clip_ivertices)[idx1].ivertex].transformed.a[collindex[1]],
    int_to_l3d_real(0),
    int_to_l3d_real(1) );

  if ( side_of_point(&ortho_point, &ortho_p1, &ortho_p2) != inside_side ) {
    status = 0;
    done = 1;
  }

  if(idx1==0) {
    done = 1;
  }else {
    idx0 = idx1;
    idx1 = next_clipidx_right(idx0, clip_ivertices->num_items);
```

```
    }
  }

  return status;
}

l3d_polygon_2d* l3d_polygon_3d_collidable::clone(void) {
  return new l3d_polygon_3d_collidable(*this);
}

l3d_polygon_3d_collidable::l3d_polygon_3d_collidable
(const l3d_polygon_3d_collidable &r)
    : l3d_polygon_3d_clippable(r),
    l3d_collidable(r)
{
}
```

Class l3d_polygon_3d_textured_lightmapped_collidable

Class `l3d_polygon_3d_textured_lightmapped_collidable` is a textured, light mapped polygon against which a collision test can be performed. It implements no new functionality; it merely inherits from multiple base classes.

Listing 8-13: `p3_cltex.h`

```
#ifndef __P3_CLTEX_H
#define __P3_CLTEX_H
#include "../../tool_os/memman.h"

#include "p3_ltex.h"
#include "p3_coll.h"

class l3d_polygon_3d_textured_lightmapped_collidable :
      virtual public l3d_polygon_3d_textured_lightmapped,
      virtual public l3d_polygon_3d_collidable
{
  public:
    l3d_polygon_3d_textured_lightmapped_collidable
    (int num_pts, l3d_surface_cache *scache);
    virtual ~l3d_polygon_3d_textured_lightmapped_collidable
    (void);

    l3d_polygon_3d_textured_lightmapped_collidable
    (const l3d_polygon_3d_textured_lightmapped_collidable &r);
    l3d_polygon_2d *clone(void);

};

#endif
```

Listing 8-14: `p3_cltex.cc`

```
#include "p3_cltex.h"

#include "../../raster/ras3_sw.h"
#include "../../tool_os/memman.h"

#define EPSILON_VECTOR 0.001

l3d_polygon_3d_textured_lightmapped_collidable::
```

```
l3d_polygon_3d_textured_lightmapped_collidable
(int num_pts, l3d_surface_cache *scache):
    l3d_polygon_2d(),
    l3d_polygon_3d(),
    l3d_polygon_3d_clippable(),
    l3d_polygon_3d_collidable(),
    l3d_polygon_3d_textured(num_pts),
    l3d_polygon_3d_textured_lightmapped(num_pts, scache)
{
}

l3d_polygon_3d_textured_lightmapped_collidable::
~l3d_polygon_3d_textured_lightmapped_collidable(void)
{
}

l3d_polygon_2d* l3d_polygon_3d_textured_lightmapped_collidable::clone(void) {
  return new l3d_polygon_3d_textured_lightmapped_collidable(*this);
}

l3d_polygon_3d_textured_lightmapped_collidable::
l3d_polygon_3d_textured_lightmapped_collidable
(const l3d_polygon_3d_textured_lightmapped_collidable &r)
    : l3d_polygon_2d(r),
    l3d_polygon_3d(r),
    l3d_polygon_3d_textured(r),
    l3d_polygon_3d_textured_lightmapped(r),
    l3d_polygon_3d_clippable(r),
    l3d_polygon_3d_collidable(r),
    l3d_texture_computer(r)
{
}
```

Class l3d_camera_collidable

Class `l3d_camera_collidable` is a camera which can collide with other objects. It inherits from both `l3d_camera` and `l3d_collidable_sphere`.

The constructor sets the radius for the collision sphere.

Overridden method `update` sets the old and new positions for the camera, which are then used during collision detection.

Listing 8-15: `camcol.h`

```
#ifndef __CAMCOL_H
#define __CAMCOL_H
#include "../tool_os/memman.h"

#include "camera.h"
#include "../dynamics/colsph.h"

class l3d_camera_collidable :
      virtual public l3d_camera,
      virtual public l3d_collidable_sphere
{
  public:

    l3d_camera_collidable(void) :
    l3d_camera()
    {collision_sphere_radius = int_to_l3d_real(5);
```

```
    };
    virtual ~l3d_camera_collidable(void) {};
    virtual int update(void) {
      l3d_collidable::old_position = VRP;
      int status=l3d_moveable::update();
      l3d_collidable::new_position = VRP;
      return status;
    }
};
#endif
```

Class l3d_world_portal_textured_lightmapped_obj_colldet

Class `l3d_world_portal_textured_lightmapped_obj_colldet` is not only the longest class name in the book, it is also a world which implements simple collision detection on the camera object, preventing the camera object from walking through walls. A crashing sound is played for every collision.

Member variable `sound_client` is a pointer to a sound client object, which knows how to interface to a sound server to play sounds.

The constructor deletes the camera object created by the base class constructor, and replaces it with a new camera of type `l3d_camera_collidable`, which can collide with the environment. It also creates a sound client and asks the sound client to establish a connection with a sound server. Currently the sound server is hard-coded to be on the local host at IP address 127.0.0.1, on the RPlay port 5556.

Overridden method `load_from_file` is essentially the same as the inherited version, except it creates polygons of type `l3d_polygon_3d_textured_lightmapped_collidable` while reading the world file.

Overridden method `update_all` implements collision detection and response between the camera and the geometry in the sector. First, we position all sector geometry into world space, then merge the polygon lists for the sector and any moveable plug-in objects it might contain. Then, we check the camera for collision against all polygons in the sector by calling method `detect_collision_with` against every polygon in the sector. The camera's inherited member variable `prevent_collision` is set to 1 during these collision checks, meaning that the camera's position is corrected so as not to collide with any polygon in the sector. Since correcting the camera's position with respect to one polygon might make the position invalid with respect to another polygon, we must again loop through all polygons whenever the camera's position has been corrected, eventually leading to a camera position not colliding with any polygon. If a collision was detected, we play a sound. Portal crossing is also newly implemented; a portal crossing is defined as a moving collision with a portal. Earlier, we explicitly searched for containment of the camera within all sectors to find out if the camera crossed into another sector. Now, we use the technique of collision detection; if the camera's line of movement crosses through a polygon portal, which we test using the ray-polygon test, we set the current sector to the sector on the other side of the crossed portal.

Listing 8-16: w_pcoll.h

```
#ifndef __W_PCOLL_H
#define __W_PCOLL_H
#include "../../tool_os/memman.h"
#include "../../sound/soundclient.h"

#include "w_porlotex.h"

class l3d_world_portal_textured_lightmapped_obj_colldet :
      public l3d_world_portal_textured_lightmapped_obj
{
  public:
    l3d_world_portal_textured_lightmapped_obj_colldet(int xsize, int ysize);
    ~l3d_world_portal_textured_lightmapped_obj_colldet(void);
    void load_from_file(const char *fname);

    void update_all(void);
    l3d_sound_client *sound_client;

};

#endif
```

Listing 8-17: w_pcoll.cc

```
#include "w_pcoll.h"
#include "../object/ocoll.h"
#include "../polygon/p_portal.h"
#include <string.h>
#include "../polygon/p3_cltex.h"
#include "../polygon/p3_cflat.h"
#include "../polygon/p_cport.h"
#include "../../raster/ras_swl.h"
#include "../../view/camcol.h"
#include "../../dynamics/plugins/plugenv.h"
#include "../../system/fact0_4.h"
#include "../../tool_os/memman.h"

#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>

void l3d_world_portal_textured_lightmapped_obj_colldet::update_all(void) {
  l3d_world_portal_textured_lightmapped_obj::update_all();
  l3d_sector *sector;
  sector = current_sector;
  l3d_polygon_3d_node *n;

  sector->reset();
  if (sector->num_xforms) {
    sector->transform(sector->modeling_xform);
  }

  l3d_polygon_3d_node *n_objects=0;
  for(int i=0; i<sector->objects->num_items; i++) {
    (*sector->objects)[i]->reset();
```

TEAMFLY

```
  if((*sector->objects)[i]->num_xforms) {
    (*sector->objects)[i]->transform
    ((*sector->objects)[i]->modeling_xform);
  }

  if((*sector->objects)[i]->nonculled_polygon_nodes) {
    l3d_polygon_3d_node *n_temp;
    n_temp = (*sector->objects)[i]->nonculled_polygon_nodes;
    while(n_temp->next) {
      n_temp->polygon->set_transform_stage(sector->transform_stage);
      n_temp=n_temp->next;
    }
    n_temp->next = n_objects;
    if(n_objects) {n_objects->prev = n_temp; }
    n_objects = (*sector->objects)[i]->nonculled_polygon_nodes;
    n_objects->prev = NULL;
  }
}

if(n_objects) {
  l3d_polygon_3d_node *n_temp;
  n_temp = sector->nonculled_polygon_nodes;
  while(n_temp->next) {
    n_temp=n_temp->next;
  }
  n_temp->next = n_objects;
  n_objects->prev = n_temp;
}

int some_collision_occurred = 1;
int do_sound = 0;

l3d_camera_collidable *cam;
cam = dynamic_cast<l3d_camera_collidable *>(camera);

if(cam) {
  l3d_point old_position = cam->old_position;

  cam->prevent_collision = 1;

  while(some_collision_occurred) {

    some_collision_occurred = 0;

    n = sector->nonculled_polygon_nodes;
    while(n) {
      l3d_polygon_3d_collidable *poly3;

      poly3 = dynamic_cast<l3d_polygon_3d_collidable *>(n->polygon);
      if(poly3 && cam &&
          !(dynamic_cast<l3d_polygon_3d_portal *>(n->polygon)))
      {

        while(cam->detect_collision_with(poly3)) {
          do_sound = 1;

          const l3d_real damping_factor = float_to_l3d_real(0.0);
          cam->VFW_velocity += l3d_mulrr(dot(cam->bounce, cam->VFW),
                                         damping_factor);
```

```
          cam->VRI_velocity += l3d_mulrr(dot(cam->bounce, cam->VRI),
                                         damping_factor);
          cam->VUP_velocity += l3d_mulrr(dot(cam->bounce, cam->VUP),
                                         damping_factor);

          cam->VRP = cam->new_position;

          some_collision_occurred = 1;
        }
      }

      n=n->next;
    }
  }

  cam->prevent_collision = 0;
  l3d_point new_pos_with_portal_displacement
  (cam->new_position.X_ + l3d_mulrr(float_to_l3d_real(5.0),
                                    cam->VFW.X_),

   cam->new_position.Y_ + l3d_mulrr(float_to_l3d_real(5.0),
                                    cam->VFW.Y_),

   cam->new_position.Z_ + l3d_mulrr(float_to_l3d_real(5.0),
                                    cam->VFW.Z_),
   int_to_l3d_real(1) ),

  original__new_pos = cam->new_position;

  n = sector->nonculled_polygon_nodes;
  while(n) {
    l3d_polygon_3d_portal_collidable *portal;

    portal = dynamic_cast<l3d_polygon_3d_portal_collidable *>(n->polygon);
    if(portal)
    {

      cam->new_position = new_pos_with_portal_displacement;

      if(cam->detect_collision_with(portal) == 1) {
        current_sector = portal->connected_sector;
        break;
      }
    }

    n=n->next;
  }

  cam->new_position = original__new_pos;

}

n = sector->nonculled_polygon_nodes;
while(n) {
  l3d_polygon_3d_collidable *poly3;

  poly3 = dynamic_cast<l3d_polygon_3d_collidable *>(n->polygon);
  if(poly3 && cam) {

    if (poly3->plane.side_of_point(cam->VRP) < 0) {
```

```
      }

    }

    n=n->next;
  }

  if(do_sound) {
    char cmd[] = "play count=1 volume=64 sound=crash.au";
    sound_client->send_command(cmd);
  }
}

l3d_world_portal_textured_lightmapped_obj_colldet::
l3d_world_portal_textured_lightmapped_obj_colldet
(int xsize, int ysize)
    : l3d_world_portal_textured_lightmapped_obj (xsize, ysize)
{
  delete camera;
  camera = new l3d_camera_collidable();

  rasterizer_3d_imp->fovx = &(camera->fovx);
  rasterizer_3d_imp->fovy = &(camera->fovy);

  sound_client = factory_manager_v_0_4.sound_client_factory->create();
  sound_client->connect("127.0.0.1 5556");

}

l3d_world_portal_textured_lightmapped_obj_colldet::
~l3d_world_portal_textured_lightmapped_obj_colldet(void)
{
  delete sound_client;
}

void l3d_world_portal_textured_lightmapped_obj_colldet::
load_from_file(const char *fname)
{
  FILE *fp;

  int linesize=1536;
  char line[1536];
  int tex_count, snum;

  fp = fopen("world.dat", "rb");

  fgets(line,sizeof(line),fp);
  sscanf(line,"%d", &tex_count);

  for(int i=0; i<tex_count; i++) {
    fgets(line,sizeof(line),fp);
    while(line[strlen(line)-1] == '') {line[strlen(line)-1] = '0'; }

    texture_loader->load(line);
    int next_tex_index;
    next_tex_index = tex_data.next_index();
    tex_data[next_tex_index].width = texture_loader->width;
    tex_data[next_tex_index].height = texture_loader->height;
    tex_data[next_tex_index].data = texture_loader->data;
```

```
}

screen->refresh_palette();

fgets(line,linesize,fp);
sscanf(line,"%d",&snum);

l3d_sector *current_sector=NULL;

for(int onum=0; onum<snum; onum++) {
  char sname[80];
  char keyword[80];
  char plugin_name[1024];
  char rest_parms[4096];
  int numv, nump;
  fgets(line, linesize, fp);
  strcpy(keyword,"");
  strcpy(sname,"");
  strcpy(plugin_name,"");
  strcpy(rest_parms,"");
  sscanf(line,"%s %s %s",keyword, sname, plugin_name);

  //- get rest parameters all in one string
  char *tok;
  tok=strtok(line, " "); //- skip keyword
  tok=strtok(NULL, " "); //- skip object name
  tok=strtok(NULL, " "); //- skip plugin name
  tok=strtok(NULL, ""); //- rest of the line until newline
  if(tok) {strncpy(rest_parms, tok, sizeof(rest_parms)); }

  if(strcmp(keyword,"SECTOR")==0) {

    fgets(line,linesize,fp);
    sscanf(line, "%d %d", &numv, &nump);

    l3d_sector *sector;
    int new_onum;
    objects[new_onum=objects.next_index()] = sector = new l3d_sector(numv);
    current_sector = sector;

    strcpy(objects[new_onum]->name, sname);

    for(int v=0; v<numv; v++) {
      int vn;
      float vx, vy, vz;

      fgets(line, linesize, fp);
      sscanf(line, "%d %f %f %f", &vn, &vx, &vy, &vz);
      (*(objects[new_onum]->vertices))[vn].original.set(float_to_l3d_real(vx),
          float_to_l3d_real(vy),
          float_to_l3d_real(vz),
          int_to_l3d_real(1));
    }

    for(int p=0; p<nump; p++) {
      int p_idx = objects[new_onum]->polygons.next_index();

      char *s;
      fgets(line, linesize, fp);
      s = strtok(line, " ");
```

```
if(strcmp(s,"GEOMPOLY")==0) {
  int num_vtx;
  l3d_polygon_3d_textured_lightmapped_collidable *pt;

  s = strtok(NULL, " ");
  sscanf(s, "%d", &num_vtx);
  objects[new_onum]->polygons[p_idx] =
    pt =
    new l3d_polygon_3d_textured_lightmapped_collidable(num_vtx,scache);
  objects[new_onum]->polygons[p_idx]->vlist =
    &objects[new_onum]->vertices;
  for(int vtx=0; vtx<num_vtx; vtx++) {
    s = strtok(NULL, " ");
    int iv;
    sscanf(s, "%d", &iv);
    (*(objects[new_onum]->polygons[p_idx]->ivertices))[
      objects[new_onum]->polygons[p_idx]->ivertices->next_index()].ivertex = iv;

  }

  pt->fovx = &(camera->fovx);
  pt->fovy = &(camera->fovy);
  pt->screen_xsize = &(screen->xsize);
  pt->screen_ysize = &(screen->ysize);

  int tex_id = 0;
  pt->texture = new l3d_texture;
  pt->texture->tex_data = & tex_data[tex_id];
  pt->texture->owns_tex_data = false;

  pt->texture->O.original.set
  (

    (**pt->vlist)[ (*pt->ivertices)[0].ivertex ].original.X_,
    (**pt->vlist)[ (*pt->ivertices)[0].ivertex ].original.Y_,
    (**pt->vlist)[ (*pt->ivertices)[0].ivertex ].original.Z_,
    int_to_l3d_real(1) );
  pt->texture->U.original.set
  (
    (**pt->vlist)[ (*pt->ivertices)[1].ivertex ].original.X_,
    (**pt->vlist)[ (*pt->ivertices)[1].ivertex ].original.Y_,
    (**pt->vlist)[ (*pt->ivertices)[1].ivertex ].original.Z_,
    int_to_l3d_real(1) );

  l3d_point V_point(
    (**pt->vlist)[ (*pt->ivertices)[pt->ivertices->num_items-1].ivertex ].original.X_,
    (**pt->vlist)[ (*pt->ivertices)[pt->ivertices->num_items-1].ivertex ].original.Y_,
    (**pt->vlist)[ (*pt->ivertices)[pt->ivertices->num_items-1].ivertex ].original.Z_,
    int_to_l3d_real(1) );
  l3d_vector
  U_vector = pt->texture->U.original - pt->texture->O.original,
             V_vector = V_point - pt->texture->O.original,
                        U_cross_V_vector = cross(U_vector, V_vector),
                                           V_vector_new = normalized(cross(U_cross_V_vector,
                                                  U_vector)) *
                                                         sqrt(dot(U_vector,U_vector));

  pt->texture->V.original = V_vector_new + pt->texture->O.original;

  objects[new_onum]->polygons[p_idx]->compute_center();
```

```
  objects[new_onum]->polygons[p_idx]->compute_sfcnormal();

  pt->compute_surface_orientation_and_size();

  pt->plane.align_with_point_normal(pt->center.original, normalized(pt->sfcnormal.original -
           pt->center.original));

}else if(strcmp(s,"GEOMPOLY_TEX")==0) {

  int num_vtx;
  l3d_polygon_3d_textured_lightmapped_collidable *pt;

  s = strtok(NULL, " ");
  sscanf(s, "%d", &num_vtx);
  objects[new_onum]->polygons[p_idx] =
    pt =
    new l3d_polygon_3d_textured_lightmapped_collidable
    (num_vtx, scache);
  objects[new_onum]->polygons[p_idx]->vlist = &objects[new_onum]->vertices;

  for(int vtx=0; vtx<num_vtx; vtx++) {
    s = strtok(NULL, " ");
    int iv;
    sscanf(s, "%d", &iv);
    (*(objects[new_onum]->polygons[p_idx]->ivertices))[
      objects[new_onum]->polygons[p_idx]->ivertices->next_index()].ivertex = iv;
  }
  s = strtok(NULL, " ");

  pt->fovx = &(camera->fovx);
  pt->fovy = &(camera->fovy);
  pt->screen_xsize = &(screen->xsize);
  pt->screen_ysize = &(screen->ysize);

  int tex_id;
  sscanf(s, "%d", &tex_id);
  pt->texture = new l3d_texture;
  pt->texture->tex_data = & tex_data[tex_id];
  pt->texture->owns_tex_data = false;
  float x,y,z;
  s = strtok(NULL, " ");
  sscanf(s,"%f", &x);
  s = strtok(NULL, " ");
  sscanf(s,"%f", &y);
  s = strtok(NULL, " ");
  sscanf(s,"%f", &z);

  pt->texture->O.original.set(float_to_l3d_real(x),
                              float_to_l3d_real(y),
                              float_to_l3d_real(z),
                              int_to_l3d_real(1));
  s = strtok(NULL, " ");
  sscanf(s,"%f", &x);
  s = strtok(NULL, " ");
  sscanf(s,"%f", &y);
  s = strtok(NULL, " ");
  sscanf(s,"%f", &z);
  pt->texture->U.original.set(float_to_l3d_real(x),
                              float_to_l3d_real(y),
```

```
                                float_to_l3d_real(z),
                                int_to_l3d_real(1));
      s = strtok(NULL, " ");
      sscanf(s,"%f", &x);
      s = strtok(NULL, " ");
      sscanf(s,"%f", &y);
      s = strtok(NULL, " ");
      sscanf(s,"%f", &z);
      pt->texture->V.original.set(float_to_l3d_real(x),
                                float_to_l3d_real(y),
                                float_to_l3d_real(z),
                                int_to_l3d_real(1));

      objects[new_onum]->polygons[p_idx]->compute_center();
      objects[new_onum]->polygons[p_idx]->compute_sfcnormal();

      pt->compute_surface_orientation_and_size();

      pt->plane.align_with_point_normal(pt->center.original, normalized(pt->sfcnormal.original -
                pt->center.original));

    }else {
      int num_vtx;

      s = strtok(NULL, " ");
      sscanf(s, "%d", &num_vtx);
      objects[new_onum]->polygons[p_idx] = new l3d_polygon_3d_portal_collidable(num_vtx);
      objects[new_onum]->polygons[p_idx]->vlist = &objects[new_onum]->vertices;
      for(int vtx=0; vtx<num_vtx; vtx++) {
        s = strtok(NULL, " ");
        int iv;
        sscanf(s, "%d", &iv);
        (*(objects[new_onum]->polygons[p_idx]->ivertices))[
          objects[new_onum]->polygons[p_idx]->ivertices->next_index()].ivertex = iv;
      }

      objects[new_onum]->polygons[p_idx]->compute_center();
      objects[new_onum]->polygons[p_idx]->compute_sfcnormal();

      s = strtok(NULL," ");
      char *s2 = s + strlen(s)-1;
      while(*s2==' ' || *s2=='') {*s2=0; s2--; }
      l3d_polygon_3d_portal *pp;
      pp = dynamic_cast<l3d_polygon_3d_portal *>(objects[new_onum]->polygons[p_idx]);
      if(pp) {strcpy(pp->connected_sector_name, s); }
    }

  }
}else if(strcmp(keyword,"ACTOR")==0) {
  int new_onum;

  objects[new_onum = objects.next_index()]
  = (*current_sector->objects)[current_sector->objects->next_index()]
    = new l3d_object_clippable_boundable_collidable(100);

  strcpy(objects[new_onum]->name, sname);
  strcpy(objects[new_onum]->plugin_name, plugin_name);
  objects[new_onum]->parent = current_sector;
```

```
    if(strlen(plugin_name)) {
      objects[new_onum]->plugin_loader =
        factory_manager_v_0_4.plugin_loader_factory->create();
      objects[new_onum]->plugin_loader->load(plugin_name);
      objects[new_onum]->plugin_constructor =
        (void (*)(l3d_object *, void *))
        objects[new_onum]->plugin_loader->find_symbol("constructor");
      objects[new_onum]->plugin_update =
        (void (*)(l3d_object *))
        objects[new_onum]->plugin_loader->find_symbol("update");
      objects[new_onum]->plugin_destructor =
        (void (*)(l3d_object *))
        objects[new_onum]->plugin_loader->find_symbol("destructor");
      objects[new_onum]->plugin_copy_data =
        (void (*)(const l3d_object *, l3d_object *))
        objects[new_onum]->plugin_loader->find_symbol("copy_data");

      l3d_plugin_environment *e = new l3d_plugin_environment
                                  (texture_loader, screen->sinfo, scache, rest_parms);

      if(objects[new_onum]->plugin_constructor) {
        (*objects[new_onum]->plugin_constructor) (objects[new_onum],e);
      }
    }
  }else if(strcmp(keyword,"CAMERA")==0) {

    float posx,posy,posz;
    float xaxis_x, xaxis_y, xaxis_z,
    yaxis_x, yaxis_y, yaxis_z,
    zaxis_x, zaxis_y, zaxis_z;
    char *tok;

    tok = strtok(rest_parms, " ");  if(tok) {sscanf(tok, "%f", &posx); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &posy); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &posz); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &xaxis_x); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &yaxis_x); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &zaxis_x); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &xaxis_y); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &yaxis_y); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &zaxis_y); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &xaxis_z); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &yaxis_z); }
    tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &zaxis_z); }

    this->current_sector = current_sector;
    camera->VRP.set(float_to_l3d_real(posx),
                    float_to_l3d_real(posy),
                    float_to_l3d_real(posz),
                    int_to_l3d_real(1));
    camera->calculate_viewing_xform();
  }
}

for(int ii=0; ii<objects.num_items; ii++) {
  for(int pi=0; pi<objects[ii]->polygons.num_items; pi++) {
    l3d_polygon_3d_portal *p;
    if ((p = dynamic_cast<l3d_polygon_3d_portal *>(objects[ii]->polygons[pi]))) {
      p->connected_sector = sector_by_name(p->connected_sector_name);
    }
```

```
    }
  }

}
```

Plug-in Object Seeker, Class l3d_plugin_videoscape_mesh_seeker

Class `l3d_plugin_videoscape_mesh_seeker` is a plug-in data class used by the plug-in `seeker.so`. The `seeker` plug-in is an object which randomly moves about the environment, giving the appearance of seeking something. Real seeking behavior for a specific target could also be programmed into the update routine.

Member variables `plugin_env`, `tex_data`, `mesh_fname`, `texcoord_fname`, and `texture_fname` are the same as with the Videoscape plug-in `vidmesh`.

Enumeration `movement_states` defines the allowable movement states for the seeker. `NONE` means no movement; the `AROUND_ states` indicate rotation around one of the seeker's local coordinate axes; and the `ALONG_ states` indicate movement in the direction of one of the seeker's local coordinate axes.

Member variable `tick_count` is a count of frames indicating how long the current movement state has been executed.

Member variable `max_ticks` is a count of frames indicating the duration of the current movement state.

Member variable `turning_velocity` represents how rapidly the object rotates about its current rotation axis, in degrees per frame.

Member variable `moving_velocity` represents how rapidly the object moves along its current movement axis, in world units per frame. Note that the use of frame-based velocities is not physically realistic; see the particle system program in Chapter 7 and the section titled "Physics" for information on more physically realistic simulation.

The plug-in constructor is identical to the `vidmesh` constructor, except it also specifies a bounding sphere for collision detection purposes.

The plug-in `update` function handles object movement and collision detection. Object movement is handled by extracting the axes of the local object coordinate system from the object's composite transformation matrix. To do this, we interpret the matrix as a coordinate system and use the first three columns of the matrix as the vectors VRI, VUP, and VFW, and the last column as VRP. Then, based on the current movement state, we multiply the current rotation matrix with a new matrix for the new rotation or translation, and save the new matrix as the composite transformation matrix. Contrast this orientation and movement method with the approach of the camera, which explicitly stored orientation as VRI, VUP, and VFW vectors. As we now see, these vectors can also be extracted from a transformation matrix, and do not need to be stored explicitly (though we might still want to for clarity).



NOTE Small numerical inaccuracies during computation mean that we should, from time to time, ensure that all of the axis vectors in the matrix are orthogonal and of length one, a process called *orthonormalization*. We can do this by normalizing VRI and VUP, computing VFW as the cross product of VRI and VUP, then recomputing VUP as the cross product of VFW and VRI. At this point all three vectors should be at right angles to one another, and should be of length one.

Collision detection in `update` is almost identical to that for the class `l3d_camera_collidable`. We check the object's bounding sphere against all polygons in the environment, and ensure that it collides with none. Additionally, we ensure that the object does not collide with any other plug-in objects in the environment. We use an `explosion_factor` variable to iteratively increase the size of the spring-like bounces which we perform after a collision is detected. In this way, multiple simultaneous collisions are eventually resolved by pushing the colliding objects apart again.

Collision detection is also used to detect when the object crosses through a portal from one sector into another. If this occurs, the object deletes itself from the previous sector's object list, and enters itself into the new sector's object list. This is so that when the new containing sector is drawn, it also correctly draws the object which just entered the sector.

The plug-in `destructor` and `copy data` functions are essentially identical to those of `vidmesh`.

Listing 8-18: `seeker.h`

```
#include "../../../dynamics/plugins/plugenv.h"
#include "../../../dynamics/colsph.h"

class l3d_texture_data;

typedef enum _movement_states
{NONE,AROUND_VFW,AROUND_VRI,AROUND_VUP,ALONG_VFW,ALONG_VRI,ALONG_VUP}
movement_states;

static const movement_states states_array[]=
  {NONE,AROUND_VFW,AROUND_VRI,AROUND_VUP,
    ALONG_VFW,ALONG_VRI,ALONG_VUP};

class l3d_plugin_videoscape_mesh_seeker
{
  public:
    l3d_plugin_environment *plugin_env;
    l3d_texture_data *tex_data;

    l3d_plugin_videoscape_mesh_seeker(l3d_plugin_environment *env);
    virtual ~l3d_plugin_videoscape_mesh_seeker(void);
    char mesh_fname[1024], texcoord_fname[1024], texture_fname[1024];

    int tick_count, max_ticks;
    movement_states movement_state;
    int turning_velocity;
    l3d_real moving_velocity;
};
```

TEAM FLY

Listing 8-19: seeker.cc

```
#include "../../../system/sys_dep.h"
#include "../../../geom/object/ocoll.h"
#include "../../../geom/object/sector.h"
#include "../../../geom/texture/texload.h"
#include "../../../geom/polygon/p3_ltex.h"
#include "../../../dynamics/plugins/plugenv.h"
#include <stdlib.h>
#include <string.h>
#include "seeker.h"

l3d_plugin_videoscape_mesh_seeker::l3d_plugin_videoscape_mesh_seeker
(l3d_plugin_environment *env)
{
  plugin_env = env;
  tex_data = NULL;
}

l3d_plugin_videoscape_mesh_seeker::~l3d_plugin_videoscape_mesh_seeker(void) {
  if(tex_data) delete tex_data;
  delete plugin_env;
}

extern "C" {

  void constructor(l3d_object *target, void *data) {
    l3d_object_clippable_boundable_collidable *obj;
    obj = dynamic_cast<l3d_object_clippable_boundable_collidable *> (target);

    obj->collision_sphere_radius = int_to_l3d_real(4);

    l3d_plugin_environment *env = (l3d_plugin_environment *)data;
    l3d_plugin_videoscape_mesh_seeker *mesh;

    target->plugin_data = mesh = new l3d_plugin_videoscape_mesh_seeker(env);
    l3d_texture_loader *l = env->texloader;

    mesh->tick_count = 0;
    mesh->max_ticks = rand() % 50 + 10;
    mesh->movement_state = states_array[rand()%7];

    char rest_parms[4096];
    strncpy(rest_parms, (char *)env->data, sizeof(rest_parms));
    char *tok;

    float posx,posy,posz;
    float xaxis_x, xaxis_y, xaxis_z,
    yaxis_x, yaxis_y, yaxis_z,
    zaxis_x, zaxis_y, zaxis_z;
    l3d_matrix position, orientation;
    strcpy(mesh->mesh_fname,"");
    strcpy(mesh->texcoord_fname,"");
    strcpy(mesh->texture_fname,"");
    position.set(int_to_l3d_real(1),
                 int_to_l3d_real(0),
                 int_to_l3d_real(0),
                 int_to_l3d_real(0),
                 int_to_l3d_real(0),
                 int_to_l3d_real(1),
                 int_to_l3d_real(0),
```

```
              int_to_l3d_real(0),
              int_to_l3d_real(0),
              int_to_l3d_real(0),
              int_to_l3d_real(1),
              int_to_l3d_real(0),
              int_to_l3d_real(0),
              int_to_l3d_real(0),
              int_to_l3d_real(0),
              int_to_l3d_real(1));
orientation = position;

tok = strtok(rest_parms, " ");  if(tok) {sscanf(tok, "%f", &posx); }
tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &posy); }
tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &posz); }
tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &xaxis_x); }
tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &yaxis_x); }
tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &zaxis_x); }
tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &xaxis_y); }
tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &yaxis_y); }
tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &zaxis_y); }
tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &xaxis_z); }
tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &yaxis_z); }
tok = strtok(NULL, " ");  if(tok) {sscanf(tok, "%f", &zaxis_z); }

tok = strtok(NULL, " ");
if(tok) {strncpy(mesh->mesh_fname, tok, sizeof(mesh->mesh_fname)); }
tok = strtok(NULL, " ");
if(tok) {strncpy(mesh->texture_fname, tok, sizeof(mesh->texture_fname)); }
tok = strtok(NULL, " ");
if(tok) {strncpy(mesh->texcoord_fname, tok, sizeof(mesh->texcoord_fname)); }

l->load(mesh->texture_fname);
mesh->tex_data = new l3d_texture_data;
mesh->tex_data->width = l->width;
mesh->tex_data->height = l->height;
mesh->tex_data->data = l->data;

FILE *fp;
FILE *fp_texcoord;
fp = fopen(mesh->mesh_fname, "rt");
fp_texcoord = fopen(mesh->texcoord_fname, "rt");
char line[4096];
if(fp) {
  fgets(line, sizeof(line), fp);
  fgets(line, sizeof(line), fp);

  int num_vert;
  sscanf(line, "%d", &num_vert);
  int i;

  delete target->vertices;
  target->vertices =
    new l3d_two_part_list<l3d_coordinate> ( num_vert );

  for(i=0; i<num_vert; i++) {
    fgets(line, sizeof(line), fp);
    float x,y,z;
    sscanf(line, "%f %f %f", &x, &y, &z);

    //- change from blender's right-handed +z-up system to a
```

```
  //- left-handed +y-up system
  (*target->vertices)[i].original.set
  (float_to_l3d_real(x),
   float_to_l3d_real(z),
   float_to_l3d_real(y),
   float_to_l3d_real(1.));
}

while(!feof(fp)) {
  fgets(line, sizeof(line), fp);
  if(feof(fp)) break;

  char *tok;

  int numv;
  tok = strtok(line, " ");
  sscanf(tok, "%d", &numv);

  l3d_polygon_3d_textured *p;
  int polygon_idx = target->polygons.next_index();
  target->polygons[polygon_idx] = p =
                                  new l3d_polygon_3d_textured(numv);
  target->polygons[polygon_idx]->vlist = &target->vertices;

  for(i=0; i<numv; i++) {
    int cur_iv=0;

    tok = strtok(NULL," ");
    if(tok) {
      int ivtx;
      sscanf(tok,"%d", &ivtx);
      cur_iv=target->polygons[polygon_idx]->ivertices->next_index();
      (*(target->polygons[polygon_idx]->ivertices)) [cur_iv].ivertex
      = ivtx;

    }

    if(fp_texcoord) {
      char texcoord_line[1024];
      float u,v;
      fgets(texcoord_line, sizeof(texcoord_line), fp_texcoord);
      sscanf(texcoord_line, "%f %f", &u,&v);

      ((l3d_polygon_ivertex_textured *)
       (&((*(target->polygons[polygon_idx]->ivertices))[cur_iv])))
      ->tex_coord.X_ = float_to_l3d_real(u);
      ((l3d_polygon_ivertex_textured *)
       (&((*(target->polygons[polygon_idx]->ivertices))[cur_iv])))
      ->tex_coord.Y_ = float_to_l3d_real(1.0 - v);
      ((l3d_polygon_ivertex_textured *)
       (&((*(target->polygons[polygon_idx]->ivertices))[cur_iv])))
      ->tex_coord.W_ = int_to_l3d_real(1);
    }
  }

  //- now reverse the list IN-PLACE; for this we need one temp swap
  //- variable, which we allocate (virtually from the list, i.e. we
  //- let the type of the temp object be allocated by the list itself)
  //- here:
```

```
l3d_polygon_ivertex *
swap_iv =
  &((*(target->polygons[polygon_idx]->ivertices))
    [target->polygons[polygon_idx]->ivertices->next_index()]);

for(i=0; i<numv/2; i++) {
  *swap_iv = (*(target->polygons[polygon_idx]->ivertices))[i];
  (*(target->polygons[polygon_idx]->ivertices))[i] =
    (*(target->polygons[polygon_idx]->ivertices))[numv-1 - i];
  (*(target->polygons[polygon_idx]->ivertices))[numv-1 - i] =
    *swap_iv;

  //- don't swap the tex coords (i.e. swap them back)

  if(fp_texcoord) {
    ((l3d_polygon_ivertex_textured *)
     ( & (*(target->polygons[polygon_idx]->ivertices))[numv-1 - i]))
    ->tex_coord =
      ((l3d_polygon_ivertex_textured *)
       ( & (*(target->polygons[polygon_idx]->ivertices))[i]))
      ->tex_coord;

    ((l3d_polygon_ivertex_textured *)
     ( & (*(target->polygons[polygon_idx]->ivertices))[i]))
    ->tex_coord =
      ((l3d_polygon_ivertex_textured *) swap_iv)->tex_coord;
  }
}

if(fp_texcoord) {

  ((l3d_polygon_ivertex_textured *)swap_iv)->tex_coord =
    ((l3d_polygon_ivertex_textured *)
     ( & (*(target->polygons[polygon_idx]->ivertices))[0] ))
    ->tex_coord;
  for(i=0; i<numv-1; i++) {

    ((l3d_polygon_ivertex_textured *)
     ( & (*(target->polygons[polygon_idx]->ivertices))[i] ))
    ->tex_coord =
      ((l3d_polygon_ivertex_textured *)
       ( & (*(target->polygons[polygon_idx]->ivertices))[i+1] ))
      ->tex_coord;
  }

  ((l3d_polygon_ivertex_textured *)
   ( & (*(target->polygons[polygon_idx]->ivertices))[i] ))->tex_coord =
     ((l3d_polygon_ivertex_textured *)swap_iv)->tex_coord;
}

target->polygons[polygon_idx]->ivertices->num_items =
  target->polygons[polygon_idx]->ivertices->num_items - 1;

p->texture = new l3d_texture;
p->texture->tex_data = mesh->tex_data;
p->texture->owns_tex_data = false;

//- no texture coord file found, so assign default tex coordinates
if(!fp_texcoord) {
```

```
    p->texture->O = (**(target->polygons[polygon_idx]->vlist))
                    [(*(target->polygons[polygon_idx]->ivertices))[0].ivertex];
    p->texture->U = (**(target->polygons[polygon_idx]->vlist))
                    [(*(target->polygons[polygon_idx]->ivertices))[1].ivertex];
    p->texture->V = (**(target->polygons[polygon_idx]->vlist))
                    [(*(target->polygons[polygon_idx]->ivertices))
                     [p->ivertices->num_items - 1].ivertex];

    p->assign_tex_coords_from_tex_space(*p->texture);
  }

  target->polygons[polygon_idx]->compute_center();
  target->polygons[polygon_idx]->compute_sfcnormal();

  p->plane.align_with_point_normal(p->center.original, normalized(p->sfcnormal.original -
              p->center.original));

}
target->num_xforms = 2;

orientation.set
(float_to_l3d_real(xaxis_x),
 float_to_l3d_real(yaxis_x),
 float_to_l3d_real(zaxis_x),
 int_to_l3d_real(0),
 float_to_l3d_real(xaxis_y),
 float_to_l3d_real(yaxis_y),
 float_to_l3d_real(zaxis_y),
 int_to_l3d_real(0),
 float_to_l3d_real(xaxis_z),
 float_to_l3d_real(yaxis_z),
 float_to_l3d_real(zaxis_z),
 int_to_l3d_real(0),
 int_to_l3d_real(0),
 int_to_l3d_real(0),
 int_to_l3d_real(0),
 int_to_l3d_real(1));

target->modeling_xforms[0] =
  orientation;

position.set(int_to_l3d_real(1),
             int_to_l3d_real(0),
             int_to_l3d_real(0),
             float_to_l3d_real(posx),
             int_to_l3d_real(0),
             int_to_l3d_real(1),
             int_to_l3d_real(0),
             float_to_l3d_real(posy),
             int_to_l3d_real(0),
             int_to_l3d_real(0),
             int_to_l3d_real(1),
             float_to_l3d_real(posz),
             int_to_l3d_real(0),
             int_to_l3d_real(0),
             int_to_l3d_real(0),
             int_to_l3d_real(1));

target->modeling_xforms[1] = position;
```

```
    target->modeling_xform = target->modeling_xforms[1]  |
                             target->modeling_xforms[0];

  }

  if(fp) fclose(fp);
  if(fp_texcoord) fclose(fp_texcoord);
}

void update(l3d_object *target) {
  l3d_object_clippable_boundable_collidable *obj;
  obj = dynamic_cast<l3d_object_clippable_boundable_collidable *> (target);

  l3d_plugin_videoscape_mesh_seeker *mesh;
  mesh = (l3d_plugin_videoscape_mesh_seeker *)target->plugin_data;

  l3d_point pos;
  l3d_vector move;

  pos.set(target->modeling_xform.a[0][3],
          target->modeling_xform.a[1][3],
          target->modeling_xform.a[2][3],
          target->modeling_xform.a[3][3]);

  if(mesh->movement_state == NONE) {
  }else if(mesh->movement_state == AROUND_VRI) {
    l3d_vector rot;
    rot.set(target->modeling_xform.a[0][0],
            target->modeling_xform.a[1][0],
            target->modeling_xform.a[2][0],
            target->modeling_xform.a[3][0]);
    target->modeling_xform =
      target->modeling_xform | l3d_mat_rotu(rot, mesh->turning_velocity);
  }else if(mesh->movement_state == AROUND_VUP) {
    l3d_vector rot;
    rot.set(target->modeling_xform.a[0][1],
            target->modeling_xform.a[1][1],
            target->modeling_xform.a[2][1],
            target->modeling_xform.a[3][1]);
    target->modeling_xform =
      target->modeling_xform | l3d_mat_rotu(rot, mesh->turning_velocity);
  }else if(mesh->movement_state == AROUND_VFW) {
    l3d_vector rot;
    rot.set(target->modeling_xform.a[0][2],
            target->modeling_xform.a[1][2],
            target->modeling_xform.a[2][2],
            target->modeling_xform.a[3][2]);
    target->modeling_xform =
      target->modeling_xform | l3d_mat_rotu(rot, mesh->turning_velocity);
  }else if(mesh->movement_state == ALONG_VFW) {
    move.set(target->modeling_xform.a[0][0],
             target->modeling_xform.a[1][0],
             target->modeling_xform.a[2][0],
             target->modeling_xform.a[3][0]);
    move.set(l3d_mulrr(mesh->moving_velocity, move.X_),
             l3d_mulrr(mesh->moving_velocity, move.Y_),
             l3d_mulrr(mesh->moving_velocity, move.Z_),
             int_to_l3d_real(0));
  }else if(mesh->movement_state == ALONG_VRI) {
    move.set(0,0,0,0);
```

```
  move.set(target->modeling_xform.a[0][1],
           target->modeling_xform.a[1][1],
           target->modeling_xform.a[2][1],
           target->modeling_xform.a[3][1]);
  move.set(l3d_mulrr(mesh->moving_velocity, move.X_),
           l3d_mulrr(mesh->moving_velocity, move.Y_),
           l3d_mulrr(mesh->moving_velocity, move.Z_),
           int_to_l3d_real(0));

}else if(mesh->movement_state == ALONG_VUP) {
  move.set(0,0,0,0);
  move.set(target->modeling_xform.a[0][2],
           target->modeling_xform.a[1][2],
           target->modeling_xform.a[2][2],
           target->modeling_xform.a[3][2]);
  move.set(l3d_mulrr(mesh->moving_velocity, move.X_),
           l3d_mulrr(mesh->moving_velocity, move.Y_),
           l3d_mulrr(mesh->moving_velocity, move.Z_),
           int_to_l3d_real(0));
}

l3d_polygon_3d_node *n;
l3d_sector *sector;
sector = dynamic_cast<l3d_sector *> (target->parent);

sector->reset();

int some_collision_occurred = 1;

pos = pos + move;
obj->new_position = pos;

obj->prevent_collision = 1;
l3d_real explosion_factor = int_to_l3d_real(0.1);

while(some_collision_occurred) {

  some_collision_occurred = 0;

  n = sector->nonculled_polygon_nodes;
  while(n) {
    l3d_polygon_3d_collidable *poly3;

    poly3 = dynamic_cast<l3d_polygon_3d_collidable *>(n->polygon);
    if(poly3 &&
        !(dynamic_cast<l3d_polygon_3d_portal *>(n->polygon)))
    {

      while(obj->detect_collision_with(poly3)) {

        pos =
          obj->new_position =
            obj->new_position + obj->bounce * explosion_factor;
        some_collision_occurred = 1;
        explosion_factor = l3d_mulri(explosion_factor,2);
      }
    }

    n=n->next;
```

```
  }

  for(int iobj=0; iobj<sector->objects->num_items; iobj++) {
    if( (*sector->objects)[iobj] != target ) {
      l3d_object_clippable_boundable_collidable *obj2;
      obj2 = dynamic_cast<l3d_object_clippable_boundable_collidable *>
             ( (*sector->objects)[iobj] );
      if(obj2) {
        if(obj->detect_collision_with(obj2)) {
          pos =
            obj->new_position =
              obj->new_position + obj->bounce * explosion_factor;
          some_collision_occurred = 1;
          explosion_factor = l3d_mulri(explosion_factor,2);

        }
      }

    }
  }
}

target->modeling_xform.a[0][3] = pos.a[0];
target->modeling_xform.a[1][3] = pos.a[1];
target->modeling_xform.a[2][3] = pos.a[2];
target->modeling_xform.a[3][3] = pos.a[3];

obj->prevent_collision = 0;

n = sector->nonculled_polygon_nodes;
while(n) {
  l3d_polygon_3d_portal_collidable *portal;

  portal = dynamic_cast<l3d_polygon_3d_portal_collidable *>(n->polygon);
  if(portal) {
    if(obj->detect_collision_with(portal) == 1) {

      printf("CROSSING PORTAL INTO %p", portal->connected_sector);

      target->parent = portal->connected_sector;

      int idx;
      for(idx=0; idx<sector->objects->num_items; idx++) {
        if((*sector->objects)[idx] == target) {
          int idx2;
          for(idx2=idx; idx2<sector->objects->num_items-1; idx2++) {
            (*sector->objects)[idx2] = (*sector->objects)[idx2+1];
          }
          (*sector->objects)[idx2] = NULL;
          break;
        }
      }
      sector->objects->num_items–;

      sector = portal->connected_sector;
      sector->objects->next_index();

      (*sector->objects)[sector->objects->num_items-1] =
        dynamic_cast<l3d_object_clippable_boundable *>(target);
```

```
        break;
      }
    }

    n=n->next;
  }

  mesh->tick_count++;
  if(mesh->tick_count > mesh->max_ticks) {
    mesh->tick_count = 0;
    mesh->max_ticks = rand()%50 + 50;
    mesh->turning_velocity = rand()%5;
    mesh->moving_velocity = l3d_mulri(float_to_l3d_real(0.5),(rand()%5)+2);

    if(rand()%2) {
      mesh->movement_state = states_array[rand()%7];
      if(mesh->movement_state == NONE ||
         mesh->movement_state == ALONG_VFW ||
         mesh->movement_state == ALONG_VRI
         )
      {
        mesh->movement_state = ALONG_VUP;
      }
    }else {
      mesh->movement_state = ALONG_VUP;
    }
  }
}

void destructor(l3d_object *target) {
  delete (l3d_plugin_videoscape_mesh_seeker *) target->plugin_data;
}

void copy_data(l3d_object *target, l3d_object *copy_target) {
  l3d_plugin_videoscape_mesh_seeker *mesh;
  mesh = (l3d_plugin_videoscape_mesh_seeker *) target->plugin_data;

  l3d_plugin_environment *new_env;
  l3d_plugin_videoscape_mesh_seeker *new_mesh;

  new_env = mesh->plugin_env->clone();
  new_env->data = mesh->plugin_env->data;
  new_mesh = new l3d_plugin_videoscape_mesh_seeker(new_env);

  strcpy(new_mesh->mesh_fname, mesh->mesh_fname);
  strcpy(new_mesh->texture_fname, mesh->texture_fname);
  strcpy(new_mesh->texcoord_fname, mesh->texcoord_fname);

  l3d_texture_loader *l = new_mesh->plugin_env->texloader;
  l->load(new_mesh->texture_fname);
  new_mesh->tex_data = new l3d_texture_data;
  new_mesh->tex_data->width = l->width;
  new_mesh->tex_data->height = l->height;
  new_mesh->tex_data->data = l->data;

  for(int i=0; i<copy_target->polygons.num_items; i++) {
    l3d_polygon_3d_textured *pt;
    pt = dynamic_cast<l3d_polygon_3d_textured *>( copy_target->polygons[i]);
    if(pt) {
      pt->texture->tex_data = new_mesh->tex_data;
```

```
        pt->texture->owns_tex_data = false;
      }
    }

    copy_target->plugin_data = (void *)new_mesh;
  }
}
```

Sample Program: collide

As is usual with l3d programs, once the supporting lower-level classes are in place, the actual application program is simple. Program `collide` creates an instance of the new world class with collision detection, then asks it to load itself from disk. This causes creation of the appropriate collidable polygons and a collidable camera. Furthermore, the world file contains references to plug-in objects of type `seeker`, which also know how to collide themselves with the environment. The main program file also plays some background music stored in audio file `bgmusic.au`.

Listing 8-20: `main.cc`, main program file for program `collide`

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../lib/geom/object/object3d.h"
#include "../lib/geom/polygon/p_flat.h"
#include "../lib/geom/polygon/p_portal.h"
#include "../lib/tool_2d/screen.h"
#include "../lib/tool_os/dispatch.h"
#include "../lib/raster/rasteriz.h"
#include "../lib/tool_2d/scrinfo.h"
#include "../lib/geom/world/world.h"
#include "../lib/system/fact0_4.h"
#include "../lib/geom/texture/texload.h"
#include "../lib/geom/texture/tl_ppm.h"
#include "../lib/pipeline/pi_lwor.h"
#include "../lib/geom/world/w_pcoll.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

main() {
  factory_manager_v_0_4.choose_factories();

  l3d_dispatcher *d;
  l3d_pipeline_world_lightmapped *p;
  l3d_world_portal_textured_lightmapped_obj_colldet w(320,240);
  void *startinfo;

  w.load_from_file("world.dat");
  w.camera->near_z = float_to_l3d_real(0.5);
  w.camera->far_z = int_to_l3d_real(5000);

  //- start background music
  char cmd[] = "play count=9999 volume=128 sound=bgmusic.au";
```

TEAM-FLY
Team-Fly®

```
    w.sound_client->send_command(cmd);

    printf("current sector %s", w.current_sector->name);
    d = factory_manager_v_0_4.dispatcher_factory->create();

    p  = new l3d_pipeline_world_lightmapped(&w);
    d->pipeline = p;
    d->event_source = w.screen;

    d->start();

    delete p;
    delete d;

    //- stop background music
    char cmd2[] = "stop bgmusic.au";
    w.sound_client->send_command(cmd2);
  }
```

Run the program as follows:

1. Change to the program directory by typing **cd $L3D/binaries/linux_x/float/app/collide** and pressing **Enter**.
2. Kill any existing instances of the RPlay sound server by typing **killall -9 rplayd** and pressing **Enter**.
3. Start the RPlay sound server in the current directory by typing **rplayd** and pressing **Enter**.
4. Type **collide** to start the program and press **Enter**.

The only reason you have to kill and restart the RPlay sound server is because the new world class specifies the unqualified filename (without a directory) `crash.au` to be played whenever a collision occurred. The program specifies the filename without a directory name because the actual directory name depends on where you unpacked the files from the CD onto your hard drive. One way around this would be for the program to call an operating system routine to determine the current directory in which it was started, and to add this directory prefix to all filenames sent to the sound server.



NOTE The background music file `bgmusic.au` was produced as follows. First, I connected the output from my electric guitar (actually, from the effects box) to the input of my sound card. Then, I used `wavrec` to start recording from the sound card, and played a riff on the guitar. Next, I used `sox` to convert the .wav file to a raw file, then used `dd` to manually trim the length of the sample so that it would loop correctly (determined by listening to the resulting raw file). With the proper length sample, I then used `cat` to place several copies of the loop into a second, longer raw file. I again ran `sox` on the raw file to add an additional echo effect of 410 milliseconds. Finally, I converted the file from raw format into the .au format, one of the many formats supported by RPlay. Ordinarily, you would probably use a graphical editing tool to do such sound editing, but I didn't have one handy; it's also possible to do it all from the command line.

After starting the program, move around as usual. Press **p**, **r** to place lamps as desired. Notice that you cannot walk through walls (in most cases; we'll discuss that in a moment). Also notice the three seeker objects that randomly wander around the environment. They also do not move through the walls or each other. The seeker objects can freely move from one sector into another

through the portals, just as you can. Note that since their movement is completely random, the seeker objects often get stuck in corners for several seconds at a time—they try to move forward directly into the corner, for instance.

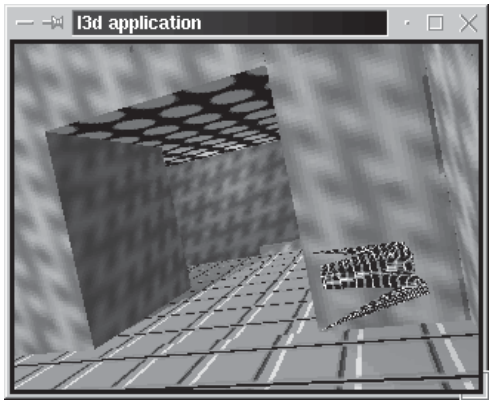


Figure 8-12: Output from sample program `collide`.

This program shows you how to combine all of the elements we have seen in this chapter and, for that matter, in this book. Let's recap the important techniques used: portals, texture mapping, light mapping, reading data from a world file, collision detection, portal crossing, and movement of plug-in objects based on their local coordinate axes.

More Advanced Collision Detection and Response

The sample program `collide` has a few shortcomings which deserve mention. Let's look at some of these, and see how we could improve the program.

First of all, the collision detection only takes place for the current sector. This means it is sometimes rarely possible for you or the seeker objects to slip outside of the room geometry by moving quickly through a portal. Since collision detection only occurs within the current sector, your new position is found to be valid with respect to all geometry in the current sector, even though it might place you outside of the geometry of the adjoining sector beyond the portal. Solving this would require us to detect all collisions with portals, and for each portal, to check collision against the connected sector's geometry as well.

The collision detection as implemented involves many redundant checks. Each object checks itself to see if it collided with any other members of the environment. This is slower than necessary because if object A already checked itself for collisions with object B, then object B no longer needs to check itself against object A. The current approach simplified the code because it allowed each object, programmed in a plug-in file, to check itself against the world. A more efficient approach introduces the concept of a higher-level collision detector, which checks for collisions among all objects in the currently active sectors, ensuring that each object pair (A,B) is checked only once, then resolves each collision independently. In other words, by moving the collision detection and response outside of the objects into a higher level, we can view the problem in the broader context of the sector and reduce the number of collision checks that must be done.

The collision detection also does not handle tunneling. An analytical sphere-sphere sweep test could detect such high-speed collisions.

The collision response as programmed is not physically realistic. If a collision is detected based on an object's movement, the object is only allowed to move as far as is possible without a collision, and an artificial bounce is added to change the object's velocity. This is unrealistic because the change in velocity is arbitrary, and because for two colliding objects, only one of them (the first one checked) gets affected by the collision. In other words, an object changes its own velocity if it notices it would have collided with another object, but in reality, both objects involved in a collision should respond to the collision. Also, all velocities were expressed in terms of change per frame, instead of the more physically realistic change per second.

Just as suggested for the collision detection system, we can also move collision response out of the domain of responsibility for each individual object, and introduce a higher-level collision response system. If we move the collision response out of the objects into a higher level in the code, this means that there must be some universal set of rules, independent of the objects involved, governing how we handle a collision between any two arbitrary objects. Fortunately, hundreds of years of scientific thinking provide us with a very practical and generally applicable set of rules to govern collisions: Newtonian dynamics. This is our next topic.

Physics

Physics is a very broad field. By providing us with mathematical models of the way the real world behaves, the field of physics allows 3D graphics programs to simulate not only the visual appearance, but also the static and dynamic properties of the real world. This can create very engaging and interesting 3D programs, because they look and act natural in subtle ways.

We already saw how a simple physical simulation can lead to impressive results with the particle system simulation, where our particles were subjected to gravity and were allowed to bounce. Let's take a look at some basic physics concepts that we use to create physically realistic simulations.

It should be mentioned at this point that we simply don't have the space here to cover physics in any depth. The purpose of this section is to give you an idea of some of the important aspects, so you can experiment and study further on your own.

Some Basic Concepts

With a few basic concepts, we can produce some physically realistic results. Some important results from Newtonian dynamics follow.

- Velocity causes an object to change position over time. Mathematically, velocity is the derivative of position over time: $v=dp/dt$. Conversely, the change in position is the integral of velocity with respect to time: $dp=\int v\,dt$. Using a simple first-order numerical integration scheme, we can say that $p'=p+v \bullet dt$. Here, p' is the new position, v is the velocity in meters/second, and dt is the elapsed time since the last frame, in seconds. This is the formula we used to compute the new positions for the particle system program.

- Acceleration causes velocity to change over time. Mathematically, acceleration is the derivative of velocity over time: $a=dv/dt$. Conversely, the change in velocity is the integral of acceleration with respect to time: $dv=\int a\,dt$. Again using a simple first-order integration scheme, we have $v'=v+a\bullet dt$. This is the formula we used to compute the new velocities for the particle system program.
- Applying forces to objects changes their acceleration. Mathematically, $F=ma$. F is the force applied to an object, m is the object's mass, and a is the object's acceleration.

In general, then, to model physics, we should model our objects as having a certain mass. By applying forces to an object, we change its acceleration. This indirectly, over time, causes a change in the velocity, which again over time causes a change in the position.

We should usually use forces to (indirectly) change the positions of objects in our world, rather than arbitrarily changing positions and velocities. There is, however, one notable exception: the contact forces that result from collisions. The problem is that with collision detection, we find the exact instant at which objects begin to touch. At this exact instant, the bodies must change direction to avoid a collision, but the correct way of doing this (applying a force) won't work, because force can only change an object's position indirectly over time. In the case of a collision, there is no time available to apply a force; the velocities must change instantaneously. We can model this with an instantaneous impulse force, which changes the velocities instantly. At the time of collision, the effective end result is a reversal of the relative velocity of the two objects. The exact change in velocity for each object depends on the objects' masses, the normal vector to the surface at the collision, and if any energy was lost during the collision. We can also take rotational effects into account. This requires us to express the angular velocity and acceleration of a body, as well the distribution of its mass.

Energy should always be conserved after any collision. No energy really gets lost in the real world; it always gets converted to another form. Therefore, we should make sure that all energy is accounted for. This again implies that we should be careful when applying forces and responding to collisions; we should base our computations on real physics equations.

Rigid Body Dynamics

The field of rigid body dynamics is generally the place to start in order to find information useful for 3D physical simulations. Dynamics is the field of physics dealing with motion and its causes; it relies on kinematics, which deals with the description of motion. Rigid body dynamics describes motion and its causes for objects whose shape or mass distribution does not change over time. For instance, a billiard ball is essentially a rigid body; when striking another billiard ball, it does not change shape. A lump of raw dough is not a rigid body; it changes shape when it strikes the floor. Rigid body dynamics can produce very realistic simulations; it follows that it is a very broad and involved field.

At this point, let's look back at the particle system program, which was the only program in this book that used real physical simulation to control motion. We mentioned that the handling of the particles was overly simplified, even though the results looked reasonably realistic. Now, with an understanding of collision detection and physics, we can understand more clearly the shortcomings of the approach taken in the particle system program.

1. The particles had neither volume nor mass, and could not collide with one another. Collision detection was performed only with the floor. Real collision detection among multiple objects would be more realistic.
2. Collision response consisted of resetting the particle's position whenever it was found to be invalid. More realistic would be a calculation of the exact collision time, either analytically or with a temporal binary search, and appropriate handling of the collision with an instantaneous impulse force at the exact time of the collision, based on the exact contact points of the collision. Collisions among multiple objects would need to be handled, by temporally sorting the collisions or by using additive spring forces.
3. The particles had no volume, no shape, and no mass. Making the objects massive would allow for realistic transfer of energy during collision response. Giving the objects shape and volume would allow for rotational effects to occur—for instance, a box spinning away if it is hit at its corner.

Other than rigid body dynamics, some other interesting areas of physical modeling include deformable objects which can be crushed or squashed, jointed objects such as humans or robot arms, cloth modeling, and objects that can disintegrate or explode in a physically realistic manner. Simulating such physical behavior is no small affair, and involves quite complex mathematics and processing power. But since processors get faster all the time, these effects will someday be as commonplace as the texture mapped polygon.

Real-Time Update and Numerical Integration

For interactive 3D applications, it is most interesting if the physical simulations occur in real time. This means that our velocity and acceleration vectors should be expressed in terms of change per unit of physical time, in other words, change per second. The particle system program illustrated how to do this by keeping track of the clock time at the last frame and subtracting it from the current clock time to determine the elapsed time.

The particle system program in Chapter 7 used a simple method of computing the new position and velocity based on the old values, called Euler integration. This method works by multiplying the previous velocity by the elapsed time to get the change in distance, and multiplying the acceleration by the change in time to get the change in velocity. This simple method can introduce errors into the computation which grow larger over time. Let's look at a simple example to understand the problem.

We'll start with the same example from the particle system. Assume that we have a particle with an initial velocity of 10m/s in the positive *y* direction, that gravity is the only force acting on the particle, and that 0.25 seconds have elapsed since the last time we called the update routine. The change in distance is the elapsed time multiplied by velocity: 10m/s*0.25s = 2.5m. The change in velocity is the elapsed time multiplied by the gravitational acceleration: 0.25s* (-9.8m/s^2)=–2.45m/s.

The problem is that we updated the position based on the velocity multiplied by the elapsed time. This multiplication assumes that the velocity was constant during the elapsed time. But, it was not constant; it was being changed continuously by acceleration caused by the force of

gravity. At the last instant we left the previous frame, the velocity was 10m/s, but this doesn't mean that in the 0.25 elapsed seconds the particle was traveling a full 10m/s the entire time. For instance, at 0.1 seconds, it was only traveling at 9.02m/s. Even at an earlier time of 0.01 seconds, it was traveling at 9.902m/s. Indeed, for any measurable time interval after the last frame, the velocity was not 10m/s because of the presence of the gravitational force, causing an acceleration. This means that by computing the change in distance since the last frame as 10m/s*0.25s, we are overestimating the distance the particle traveled (overestimating only in this particular case; in general, we can over- or underestimate). Such an error adds up over time, causing so-called *numerical integrator drift*. Fundamentally, the problem is that we have quantities such as position, velocity, and acceleration, that change continuously over time, but we are only discretely approximating the values with constant values. In actuality, we are integrating the quantities, which are differential equations, over time. This is a concept from calculus, which effectively gives the area underneath the curve of the integrated quantity (this is the geometric interpretation of an integral). We can't circumvent the discreteness of the computation, but we could use a *higher order* integration scheme, where the continuous curved change of the time-dependent quantity within the discrete time interval is also taken into account.

All integration schemes suffer from some degree of instability, but Euler integration is particularly sensitive. If the time intervals between successive frames are too large, then the error introduced by assuming a linear motion during the time interval also becomes large. This is the same phenomenon which causes tunneling effects. With physics routines, this can cause objects to be at positions they shouldn't be. If the position of an object influences other objects, then these other objects will also be at the wrong positions, with wrong velocities and accelerations. Eventually, the cumulative error grows too large, and the entire system becomes unstable and behaves in an unrealistic manner.



NOTE The sensitivity on the size of the time step leads to the notion of a *stability condition* which must be satisfied for the numerical integration to converge to the true solution. In one numerical weather prediction project with which I was involved, this stability condition was called the Courant-Friedrichs-Lewy (CFL) stability condition, in recognition of some of the earliest scientists who grappled with these problems. This numerical weather prediction package also had to integrate quantities over time; if the time step was too large, the whole solution became unstable.

Artificial Intelligence

One exciting area which can also make 3D applications more believable is the field of artificial intelligence. Artificial intelligence (AI) can be seen as the study of how to make computers respond in a way which seems intelligent to a human viewer. In the context of interactive 3D applications, especially games, this means creating entities within the virtual world that appear to be alive and appear to act in a way that a human might act. For instance, the seeker objects in the sample program `collide` were not intelligent. They randomly rotated and moved around, even trying to move ahead straight into a wall. A human would probably not do this.

In games, an important part of AI is path finding. Objects need to move from one area of the world into another to accomplish certain goals. Research from the area of robotics has led to results in this area which can be equally well applied to 3D programs. We already saw one path finding routine when we discussed camera tracking. In fact, camera tracking can be considered a simple form of AI. The camera has to decide for itself how to keep the subject in sight. Sometimes, the path finding problem is reduced from 3D to 2D; in other words, the objects are restricted to finding paths in 2D rather than in 3D. This can make algorithms simpler, since 2D path finding is essentially a 2D maze-solving problem. *Artificial Intelligence, Second Edition* by Elaine Rich and Kevin Knight describes two ways of performing path finding: the use of potential fields, where an object is attracted to its target and repelled from obstacles, and the use of visibility graphs, where every vertex of all geometry within the scene is connected to all other vertices visible from that vertex [RICH91]. Visibility between two vertices also means that a free path exists between those two vertices; thus, the shortest path in the visibility graph from the object to the target is traversed. Since the object is probably surrounded by a bounding sphere, it cannot travel all the way up to each point in the visibility graph (because it collides with the polygon first), so we can artificially displace all vertices from their polygons by the radius of the bounding sphere when forming the visibility graph. This forms a *configuration space*, which is a modified version of the actual underlying space used for path finding.

AI also often involves constructing graphs and searching these graphs for one solution or for the optimal solution to a problem. Visibility graphs are one example of this; the optimal solution in this case is the shortest path, but any solution that leads to the target is also correct. Graph searching can be done in typical exhaustive depth-first or breadth-first manner. A better technique from the field of AI is the "best first search," in which the potential cost of each of the next possible nodes is evaluated, and the most promising, least cost node is selected next. If searching one branch becomes less and less promising, the search for that branch is temporarily suspended as other, more promising paths are explored. We might even end up coming back to the originally suspended branch, because it might again become more promising than all other branches. This is a simplification of the well-known (in the field of AI) algorithm called the "A* algorithm."

Some non-procedural languages can be used for more natural expression and implementation of goal-directed behavior. Examples include the languages PROLOG and LISP, both of which have been used widely in the field of AI. Interestingly, Makefiles are also non-procedural in a very similar way; the goal is to compile the target program, and the inference rules are the dependencies we provide. Goal-directed AI programming can take a similar form, where the computer might try to reason why you have hidden for several seconds behind a wall without firing. Maybe you have no ammunition, or maybe you are planning an ambush. We can program rules in a non-procedural language to allow the computer to draw conclusions based on its observations.

AI can also mean introducing imperfection into computer behavior. A human does not always act predictably, or always hit the target. By introducing some random elements into the logic, and allowing for some autonomous decision making, behavior can appear more naturally intelligent.

Using a state machine and having a stock of predefined behaviors to choose from is another useful technique. For instance, if we are modeling the behavior of a dog, he might have the states `SLEEPING`, `HUNGRY`, `ANGRY`, `FRIENDLY`, and so forth. There would be definite transitions

from state to state depending on external inputs—the passage of time, the actions of the player and other objects, and so forth. Then, depending on the state, the dog would respond differently to its surroundings. If you walk past a dog in the `FRIENDLY` state, he might jump up in greeting, or do nothing. If you walk past a dog in the `ANGRY` state, he might start to chase you, or bite you. Even if there are no inputs from the environment, the state can still control behavior to give the appearance that the object is still active and thinking.

Finally, AI doesn't have to be complex to be effective. Try observing behavior, asking why you would or would not do something that way, and then try to program the correct behavior. For instance, take the seeker objects from the `collide` program. Why do they appear unintelligent? They have no goal. Also, their random rotations appear odd. So, a first pass at making them more intelligent would be to give them some goals—for instance, stay within a certain distance of the player. We could also reduce the probability of random rotations, and always ensure that the object returns to an upright orientation within, say, 10 seconds. This would make the objects "prefer" to remain upright, much as a human will tend to prefer viewing a scene upright.

AI, like physics, is a vast field. As you can guess, we've only barely scratched the surface here.

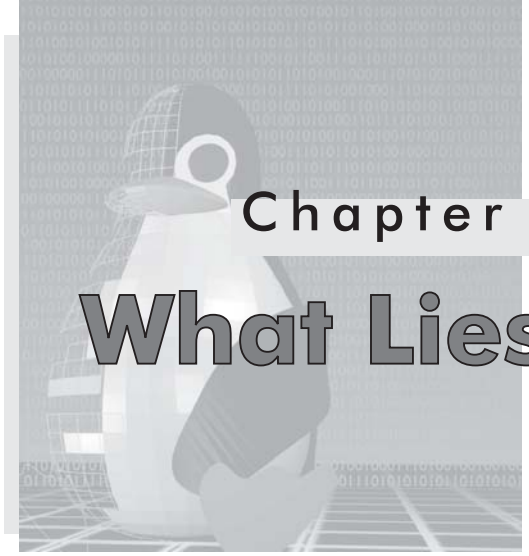
Summary

This chapter discussed some non-graphical techniques useful for interactive 3D programs. In particular, we looked at:

- Digital sound and the RPlay sound server
- TCP/IP networking with sockets
- Collision detection
- Physics
- Artificial intelligence

Using these techniques can help 3D programs become more immersive and engaging 3D environments.

One last chapter remains in this book. In it, we speculate about the future of Linux 3D graphics programming, including a peek at two high-level 3D content development systems for Linux which can greatly reduce the amount of programming needed to get a complex 3D application, such as a game, up and running. Does this mean that everything you've read in this book can be forgotten, because the content development system does all the work for you? Read on, and find out, in the exciting and long-awaited conclusion to our journey through Linux and 3D graphics programming.



Chapter 9
What Lies Ahead?

This chapter aims to give you a glance at some exciting trends affecting Linux 3D graphics programming. We begin by looking at two *content development systems* currently available for Linux. We then make some speculations about future directions of 3D graphics. Finally, we review the contents of this book within the context of 3D graphics programming in general.

Content Development Systems

In this section, we'll take a brief look at two 3D content development systems. Here, the term *3D content development system* refers to a visual 3D environment that allows the creation of interactive content with a minimum of programming. Such systems show us the direction that future 3D development might take.

Game Blender/Blender 2.0

Blender 2.0, also called Game Blender, is a version of Blender that includes some support for creating interactive content. We don't have the space in this book to go over the complete details of the entire system. What we can do is go over a quick example of how to create a simple game in Blender, so that you can see how the system works and experiment further on your own.



NOTE To execute the following tutorial, you must download the Game Blender system (version 2.0 or higher) from the Blender home page, `http://www.blender.nl`. The program is free of charge.

The following tutorial shows you how to create a simple world in Blender and navigate through it in real time.

1. Start Blender and clear the screen by pressing **Ctrl+x, Enter**. Select and delete the default plane object. Ensure you are in top view: press **7** on the numeric keypad.
2. Add a large cube object by selecting **Add**, **Mesh**, **Cube** from the Toolbox (press **Space**). Scale the cube so that it is approximately 15 units long on each side. Exit EditMode.

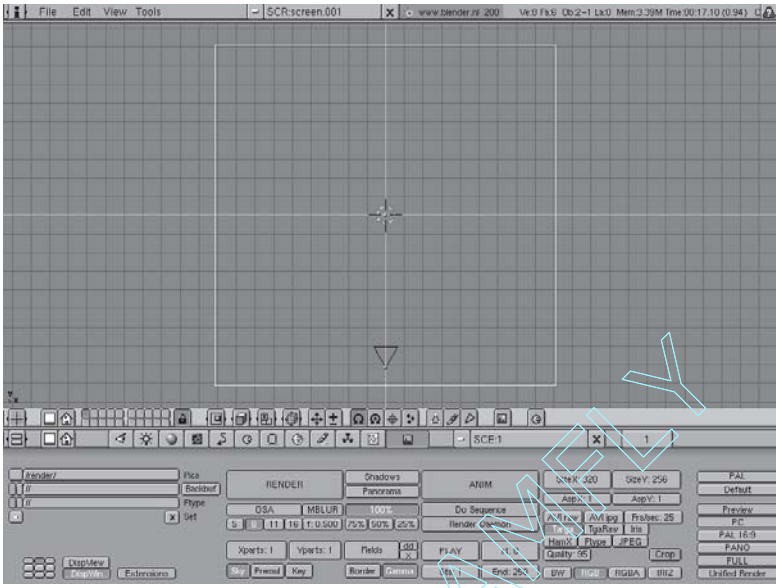


Figure 9-1

3. Change to the RealtimeButtons by pressing **F8**. Click the **Sector** button in the upper left of the ButtonsWindow. This sets the large cube mesh to be a sector.



NOTE Starting with version 2.11, Blender has done away with explicit marking of sectors, so you do not need to execute the previous step with Blender 2.11 or higher.

4. In the 3DWindow, add an Icosphere by selecting **Add**, **Mesh**, **Icosphere** from the Toolbox. Select subdivision 2, and press **Enter**. Drag the topmost vertex of the sphere slightly forward, so that the sphere has a discernible tip. Exit EditMode. Ensure the sphere is still selected.

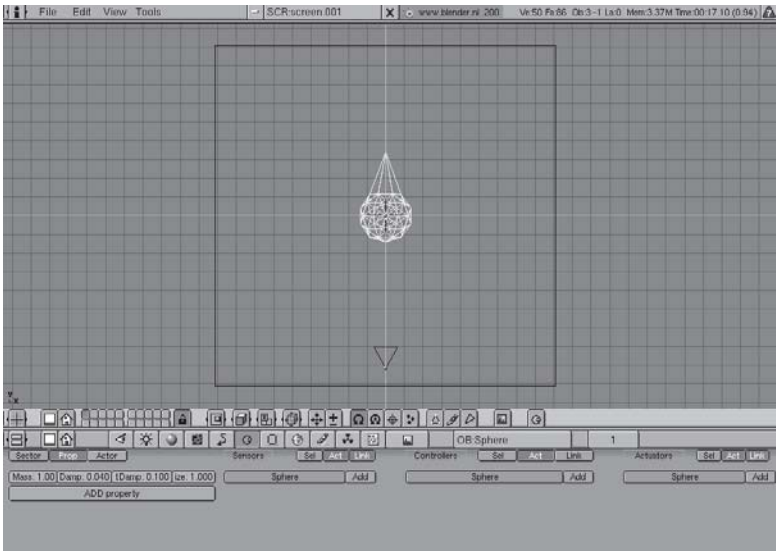


Figure 9-2

TEAM-FLY

5. In the RealtimeButtons, click the **Actor** button to flag the sphere as an actor. Then click the **MainActor** button, which marks the sphere as the main actor in the game. Also click the **Dynamic** button, which allows the object to be controlled by the Blender dynamics system.



NOTE Blender 2.11 has a new interface for the rigid body dynamics subsystem. There is no MainActor button. Also, click the RigidBody button to allow the sphere to be controlled as a rigid body by the physics subsystem.

6. Move the cursor into the ButtonsWindow and maximize it by pressing **Ctrl+Down Arrow**. We are now going to program in the keyboard handler so that we can move the sphere around with the keyboard. Doing this requires us to create sensors, controllers, and actuators.
7. In the Sensors column click the **Add** button four times. Do the same in the Controllers and Actuators columns. This creates four sensors, four controllers, and four actuators. We are going to have four keys which we use to control the sphere, and thus, four objects of each type.

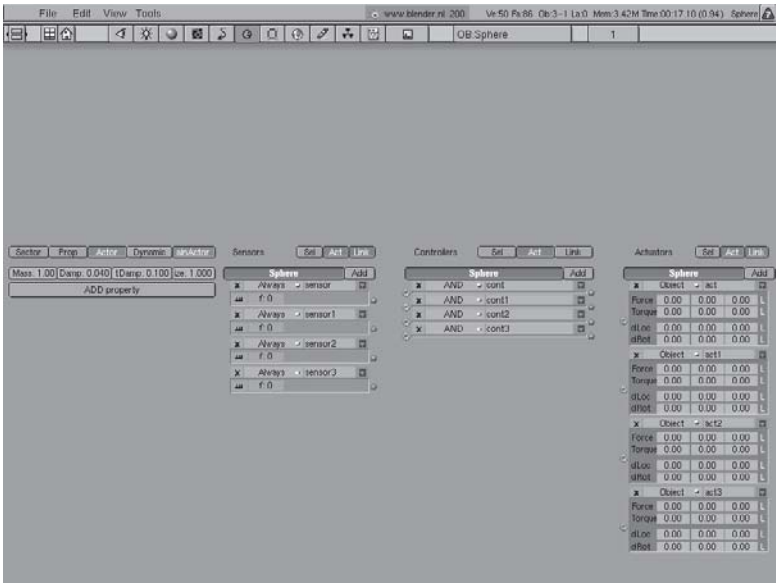


Figure 9-3

8. Notice the small pebble to the right of each sensor, and the small hole to the left of each controller. Left-click the pebble and drag it into the hole, creating a link between sensor and controller. Do this for all controllers. Also repeat the procedure with the controllers and the actuators.

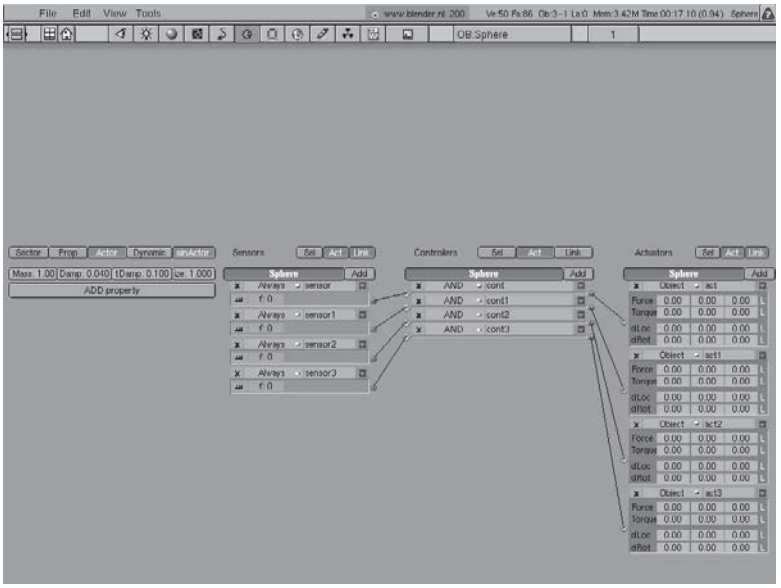


Figure 9-4

9. Notice that each sensor contains a pop-up menu field with the value "Always." Change each of these fields to contain the value "Keyboard": Click on each field, notice the menu which appears, and select **Keyboard**.

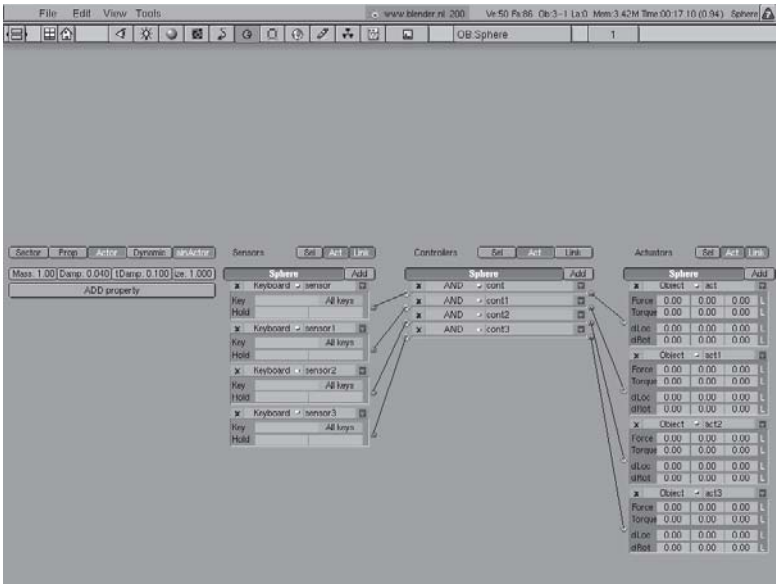


Figure 9-5

10. Assign each keyboard sensor a different key as follows. Click on the Key field for the first sensor. Notice the message "Press any key." Press **Left Arrow**. Notice that "Leftarrow"

appears in the field. Do the same for the remaining three sensors, with keys **Right Arrow**, **Up Arrow**, and **Down Arrow**.

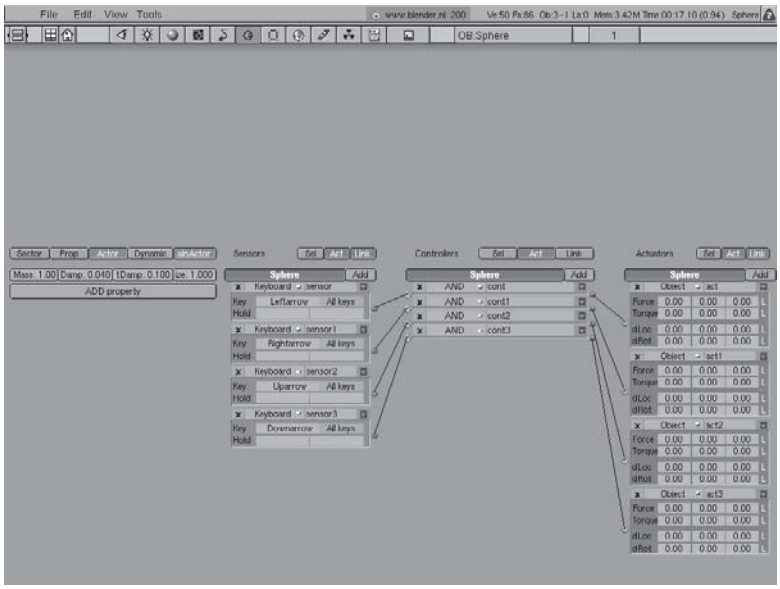


Figure 9-6

11. Change the actuators to respond to the keys as follows. For the first actuator, type **1.00** in the third torque field. For the second actuator, type **-1.00** in the third torque field. For the third actuator, type **1.00** in the second force field. For the fourth actuator, type **-1.00** in the second force field. The fields in the actuators allow you to apply forces and torques to the object whenever the sensor fires. The three fields represent *x*, *y*, and *z* components of the applied force.

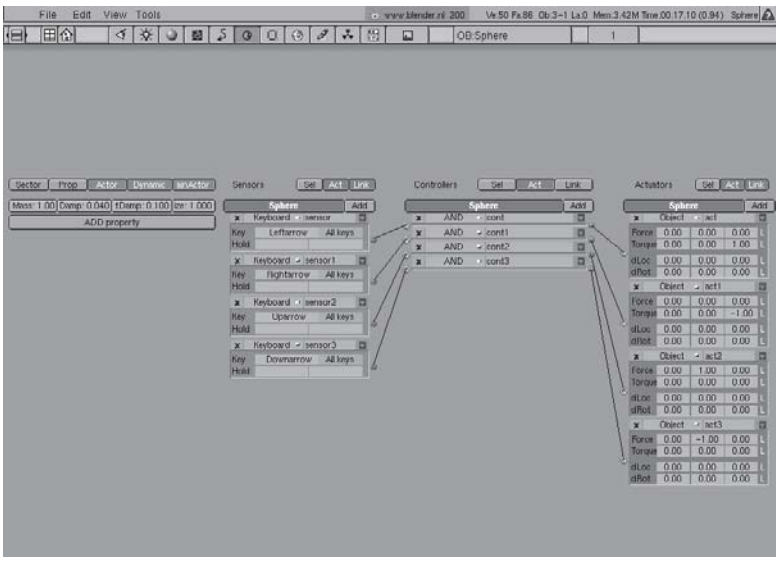


Figure 9-7

12. Restore the ButtonsWindow to its original size by pressing **Ctrl+Down Arrow**.
13. In the 3DWindow, add another cube object, slightly smaller than the original cube. Flip all vertex normals to point inside by typing **a** until all vertices are selected (yellow), then press **Ctrl+Shift+n** and press **Enter** to flip the normals. Exit EditMode. Ensure the cube object is still selected.

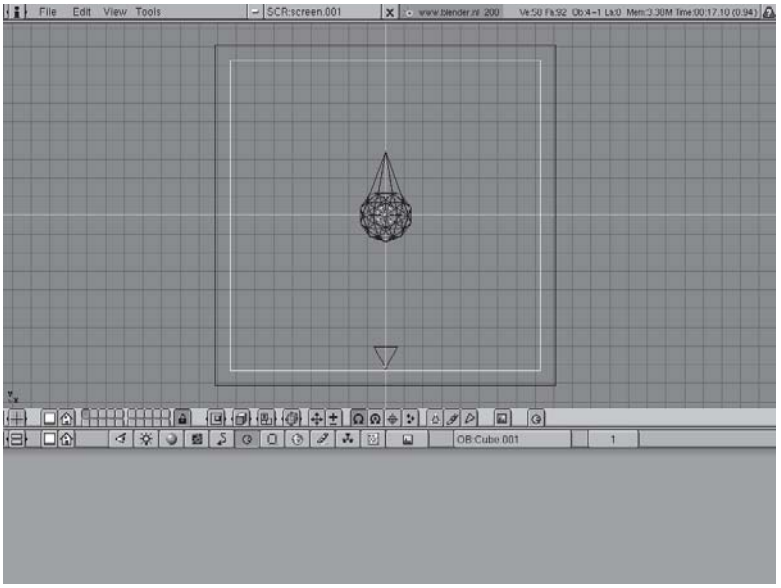


Figure 9-8

14. Move the mouse cursor over the ButtonsWindow, and press **Home**. This ensures that all buttons are visible. Then, click the button **Prop**. This marks the smaller cube as an object within the larger sector.



NOTE Blender 2.11 has eliminated the Prop button; props do not need to be explicitly marked.

15. In the 3DWindow, select the camera object by right-clicking. In the RealtimeButtons, click **Actor**. Add a sensor, controller, and actuator, and link them together. Leave the sensor type with its default value "Always."
16. Change the actuator to be of type "Camera": Click on the pop-up menu that reads "Object," and notice the menu which appears. Select **Camera**. This type of actuator causes the camera to follow a target object.
17. In the actuator field OB:, type **Sphere**. This is the name of the Icosphere object we added earlier, and is the object that the camera will track. In field Height:, type **5.00**. In field Min:, type **5.00**. In field Max:, type **10.00**. These set the constraints on the camera position with respect to the target object.

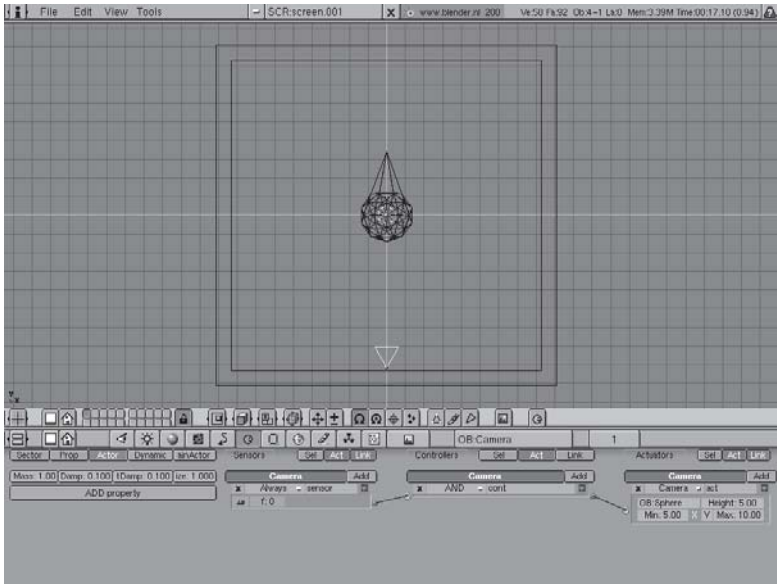


Figure 9-9

18. In the 3DWindow, switch to camera view by pressing **0** on the numeric keypad. Press **Shift+p** to start playing the game.
19. Press the arrow keys to move the sphere through the environment. Notice that the camera tracks the object automatically, that you cannot move through the walls or floor of the environment, and that gravity forces are applied. Press **Esc** to end the simulation.

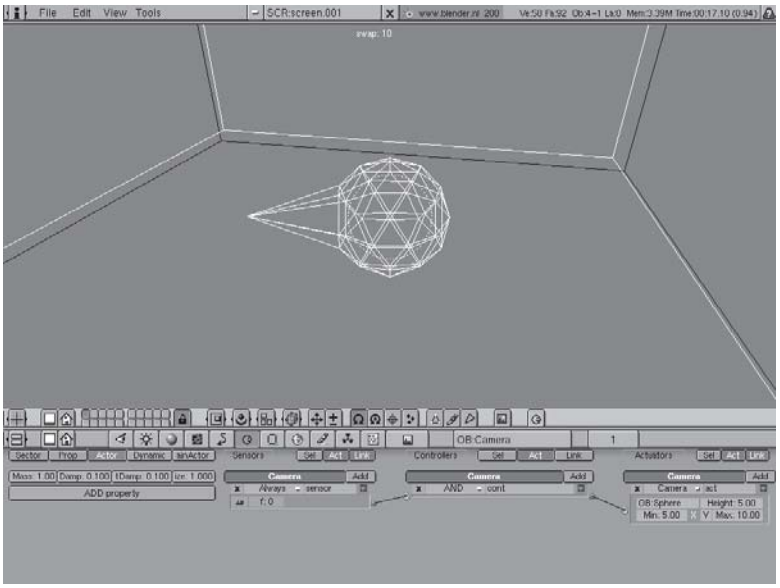


Figure 9-10



NOTE In Blender 2.11 the sphere will bounce up and down upon hitting the ground, because of the default collision handling parameters. You can correct this by explicitly specifying the material properties of the sphere and/or the surrounding cube through the MaterialButtons.



TIP You can also run the simulation from any other viewpoint, not just camera view. For instance, top view is useful for getting a bird's-eye picture of the environment.

A few notes on the system are in order:

- You can run the game in textured mode (press **Alt+z**), but this requires a hardware accelerator for acceptable performance.
- Python scripting support is planned to allow for game logic to be coded in the Python language. Preliminary support already exists.
- The visibility scheme is apparently a modified portal scheme where the portals are the faces of the bounding boxes, as discussed in Chapter 8.

This simple example gives you an idea of how the Game Blender system works. The system is very visual, using links connected via drag-and-drop in the RealtimeButtons. A system that achieves similar goals but which takes a different approach is offered by World Foundry, which we look at next.

World Foundry

World Foundry is an open source (GPL) package that enables non-programmers to create 3D environments and games. It combines a powerful 3D engine with an external level editing system and an internal scripting system.



NOTE The World Foundry project (`http://www.worldfoundry.org`) currently needs interested 3D programmers to assist in porting and extending the engine and tools. You—yes, you!—can help in this process. World Foundry is an extremely powerful system; I have not seen any other open source projects which rival it. Furthermore, World Foundry is cross platform, meaning that games can be developed for Linux and Windows.

The Blender system for assigning behaviors to objects involved visually dragging links to connect sensors, controllers, and actuators. In contrast to this, the World Foundry approach uses scripts to control behavior. The scripting language is a dialect of Scheme, which is a variant of LISP. World Foundry provides a number of example scripts for common behaviors. Creation and placement of geometry in 3D space within World Foundry is still done visually, in a 3D level editing program. (Indeed, I am attempting to adapt Blender to be used for exactly this purpose.)

One can argue whether scripting or a visual system is more flexible. My personal opinion as a programmer is that for real reuse and modularity with complex structures, there is no substitute for using a programming language. Formal languages allow us to express and manage complexity in a precise way. Visual development tools usually are good for rapid development, but are poor when it comes to reusable structure. It could be argued, however, that the types of structures created with a visual system are inherently simple, because the underlying system hides the complexity. You'll have to decide for yourself which system best suits your needs, your style, and your project. It

should be stressed, though, that neither Blender nor World Foundry use exclusively one paradigm or the other; each tool combines both visual and scripting elements in its own unique way to enable creation of content.

Let's briefly mention the key features of the World Foundry system, then go over the creation of a sample level in World Foundry.



NOTE The material for this section was adapted, with permission, from the World Foundry reference manual.

The features of the World Foundry system include:

- A variety of core game objects such as moving and static platforms, monetary units, projectiles and other weapons, shadows, enemies, object generators, shields, activation boxes, vehicles, cameras, main characters, particle systems, and so forth
- Animated textures for video walls or organic models such as lava and clouds
- Level of detail support including billboards and reduced polygon models
- Streamed textures and/or models allowing very complicated worlds to fit within small amounts of memory
- User controllable physics parameters and three-tier collision detection
- Paths to constrain object movement to a predefined course of motion (multiple movement handlers, including path, physics, stationary, and script controlled)
- Scripting language for controlling object behavior; inter-object communication allows objects to query the status of other objects and of the level
- User controllable camera tracking with a sophisticated Director/Camera Shot model
- Cross platform development of games, with main targets being Windows and Linux

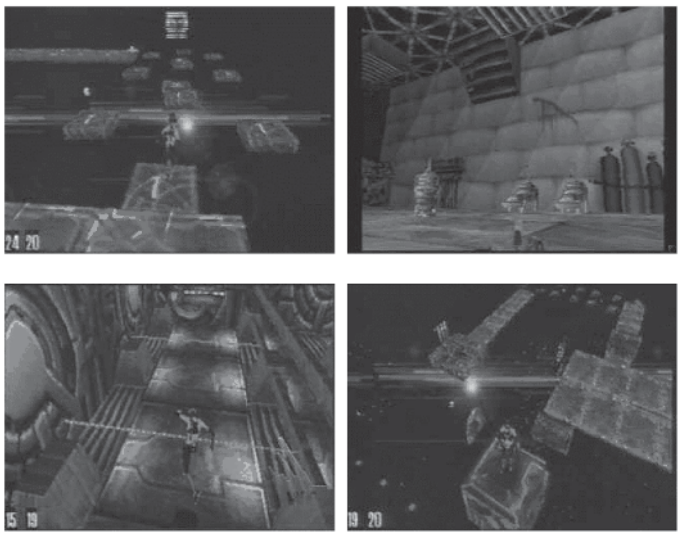


Figure 9-11: Some of the types of worlds possible with World Foundry.

Now, let's look at the creation of a simple level in World Foundry.



NOTE The instructions presented here were written for this book by the primary World Foundry programmer and architect, Kevin Seghetti, and are based on the Windows version of World Foundry, which currently uses 3D Studio Max as the level editor. Although the World Foundry engine already runs on Linux, the development tools are still being ported; I am attempting to help with the level editing portion of the conversion process, and am currently focusing on attempting to use Blender for this purpose, though a long-term solution will probably use a different and open source 3D modeling package. Therefore, the instructions here might change slightly in the final level editing system. The goal here is to show you fundamentally how level editing works in World Foundry.

This assumes you already have a working knowledge of 3D Studio Max.

1. Pull down the Preferences menu, and select **System Unit Scale**. Check the **Metric** check box, and make sure that Meters appears in the pull-down box. This is important; none of your levels will work correctly unless you perform this step (only required once after 3D Studio Max is installed).
2. Pull down the Views menu, and select **Units Set up**. Check the **Metric** check box, and make sure that Meters appears in the pull-down box. You can use any units you like, but this tutorial uses meters.
3. Start by creating a floor, to prevent other things from falling out of the world. Create a box and name it Floor01. An example size to use is 10 meters wide and half a meter thick.
4. Select the World Foundry Attributes plug-in by clicking on the **Utilities** tab (looks like a hammer) in Max, then selecting the **Attributes** plug-in (you might need to click the More button and find it in the list).
5. Ensuring the floor object is still activated, make it a StatPlat object by pulling down the drop-down menu just under the Copy and Paste buttons. (Objects default to Disabled, which means they get ignored by the engine.)

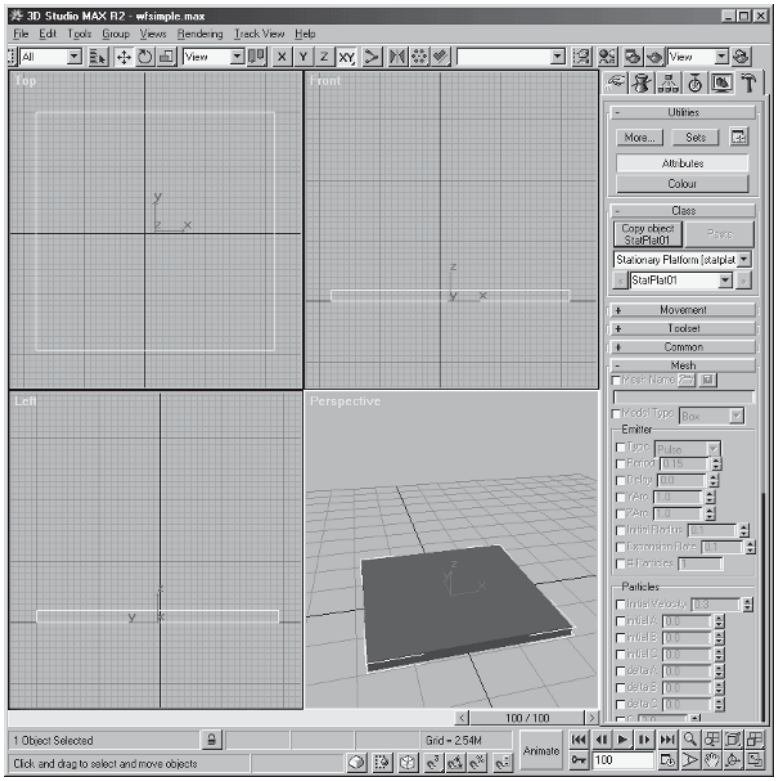


Figure 9-12: Floor with class set to StatPlat.

6. This level will have a single room that contains all of the objects. Create a box called Room01 which contains the floor and ample space above it (for instance, 6 meters). Assign it the Room class with the Attributes plug-in. To make it easier to see inside of the room I always set the room object to a material which has an opacity of 10% or so: select the room object, click on the material editor button, set the opacity to 10, and click the **Assign material to selection** button.

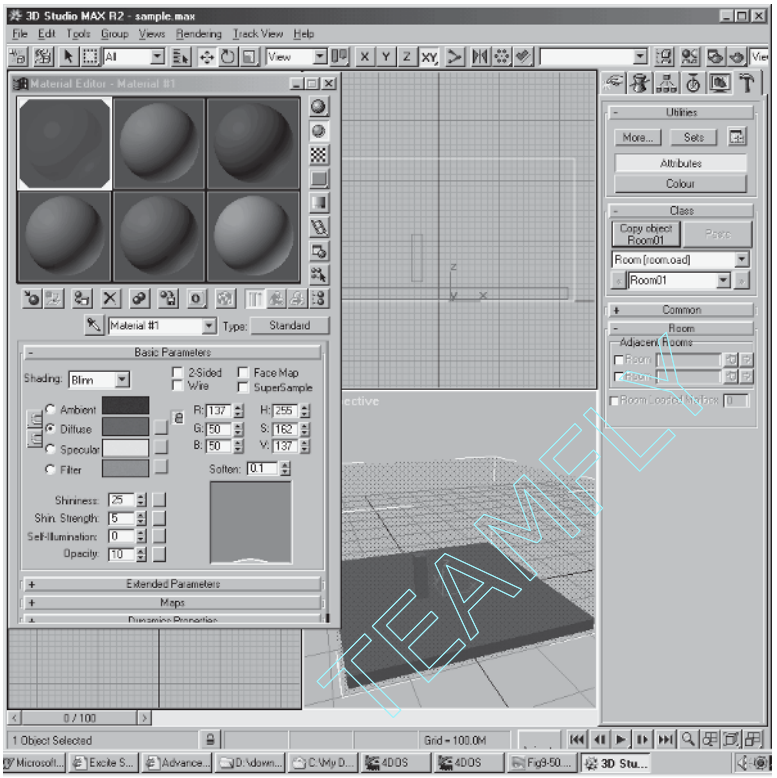


Figure 9-13: Room object, set to class Room, with opacity set to 10%.

7. Now add a player-controlled character. Anywhere within the room and above the floor, create a box of roughly humanoid size (half a meter wide and two meters tall), name it Player01, and assign it the Player class. In order to receive and process joystick messages, our Player object will need a simple input-handling script. This would look as follows.

```
(
include objects.id
include ../mailbox.def
include ../joystick.def
include ../user.def
( write-mailbox self INPUT ( read-mailbox self HARDWARE_JOYSTICK1 ) )
)
```

Enter this script into the player as follows: select the player, then in the Attributes editor open the Common roll up, click the check box left of Script to enable it (from here forward always turn on the left check box first to enable the field being edited), then click on the **New** button on the right. This will open a text editor; type or paste in the above script. Then close the editor (it will ask if you want to save your changes; select **Yes**). Finally, turn on the Script Controls Input check box just below Script.

Team-Fly®

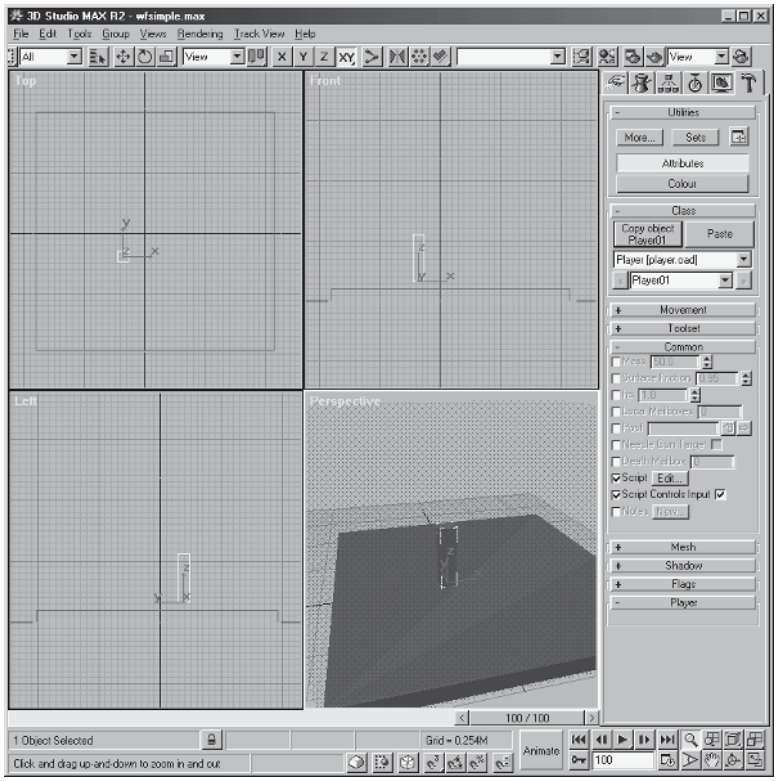


Figure 9-14: Player object with class set to Player.

8. Now we need to set up a camera shot so we can see our room. Create a small box somewhere in the room (its size and location aren't important for this example), and name it Camshot01. Assign it the Camshot class.
9. We need to create a special object to tell the Camshot where its subject is (where to look and where to focus). Create a small box a few meters away from Camshot01, name it Target01, and assign it the Target class. The distance and direction between the Camshot object and the Target object define the camera-to-subject spatial relationship. We also need to tell the Camshot which object in the levels is its target, so select Camshot01 and set Camshot—Look At to Target01. Do the same for Camshot—Follow. Then set Camshot—Track Object to Player01. This will set up a simple static camera shot.

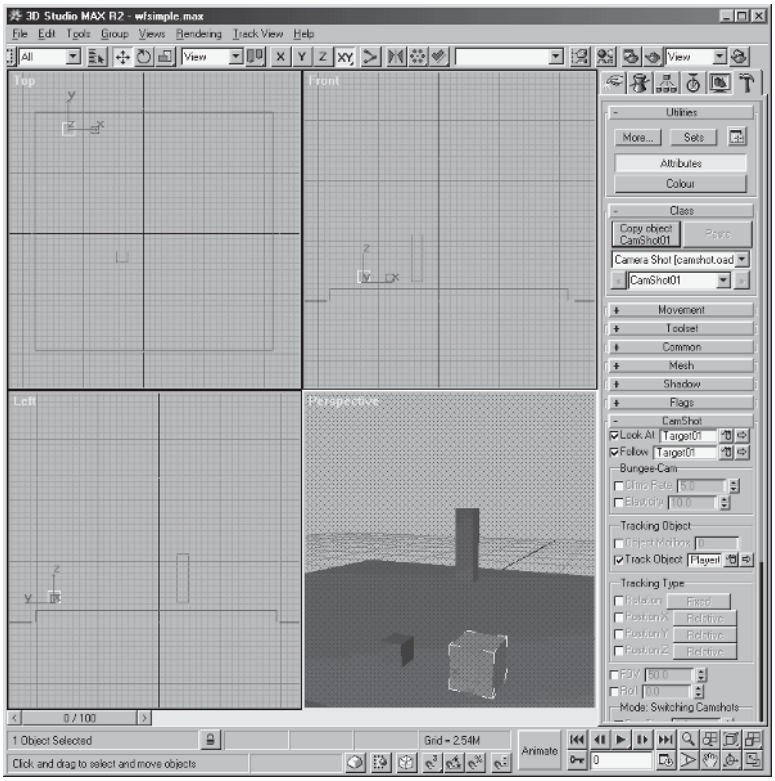


Figure 9-15: Camshot with target and tracking fields set up. On your screen the purple object is the Camshot, the red object is the target, and the blue object is the player.

10. Camera shots are activated by a special kind of Activation Box object, called an ActboxOR (short for Activation Box with Object Reference). Create a box that encloses the floor and player objects, but is itself contained by the Room box. Name it Cambox01 and assign it the ActboxOR class. We want this activation box to activate Camshot01 any time that Player01 is inside of Cambox01, so set ActBoxOR—Mailbox to 1021 (the mailbox to write CamShot references to; see reference manual for details), Object to Camshot01, Activated By to Actor, and Actor to Player01.

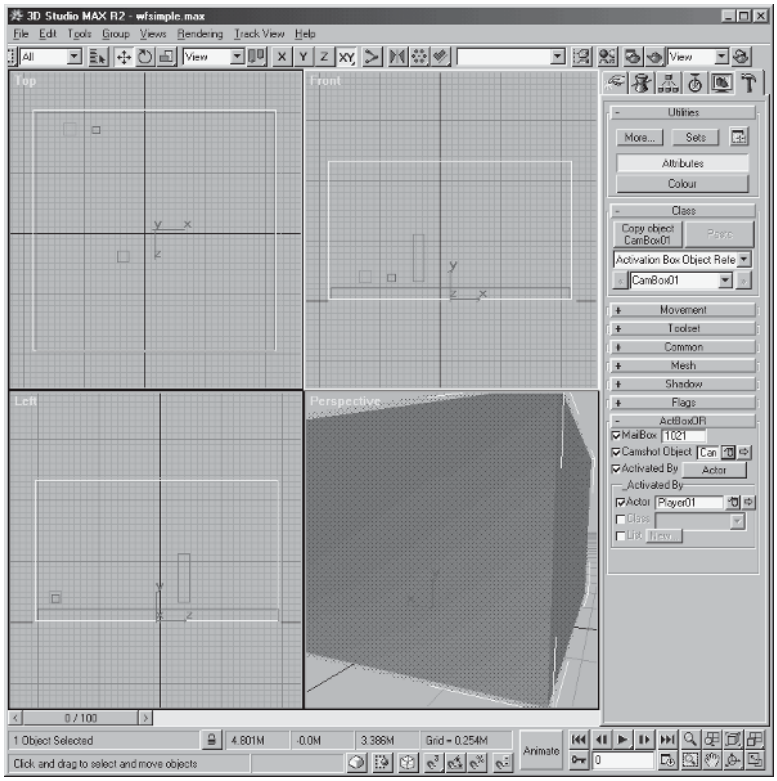


Figure 9-16: Camera Activation Box configured to set the camera to CamShot01 whenever the player is inside of it.

11. Also set this object to have a low opacity via the materials editor. Another good trick is to hide the room and camera act box to make it easier to see the objects inside of the room: click on the display tab (looks like a monitor), select the room object, click **hide selected**, select the camera act box, click **hide selected**.
12. Finally, we need to create a few objects which must exist somewhere in the level. (Their actual position is unimportant, as is their size; they hold data that the game engine will need when you run the level.) Create a small box somewhere in the Room, name it Camera01, and assign it the Camera class. Create another one named Levelobj and assign it the LevelObj class. Create a third called Matte01 and assign it the Matte class.

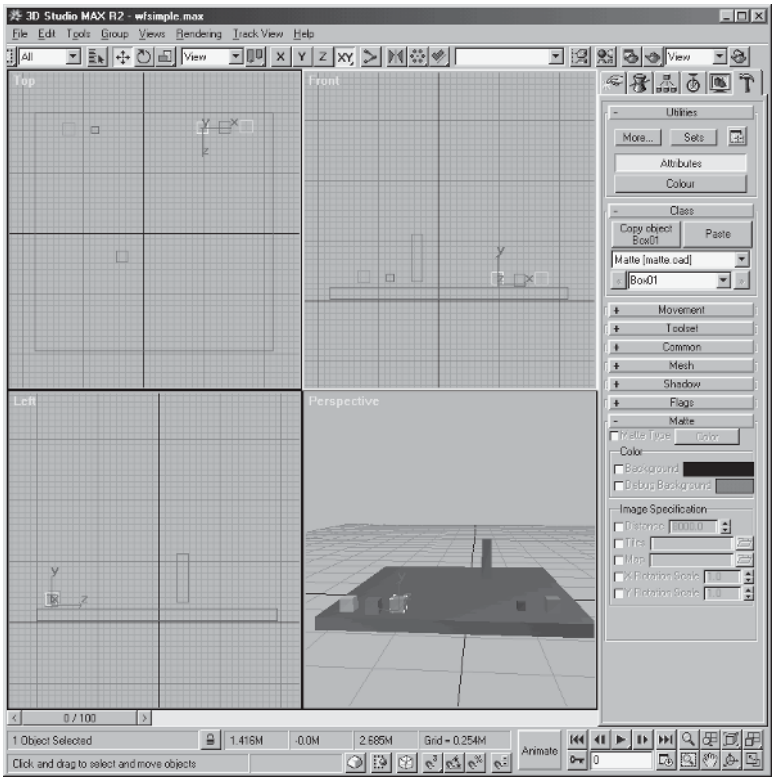


Figure 9-17: Camera, LevelObj, and Matte classes added.

13. That's all. Save your work, and export it as a World Foundry LVL file using Export from the File menu.

As you can see, creating levels with World Foundry is also done by using a level editing tool to place objects in 3D space. The main difference with the Blender system is that linkage and control of game objects occurs through scripting rather than through lines and fields on the display.

World Foundry has plans for adding wizards that automate common level-building tasks. Blender has plans for allowing Python scripted control of the game world. In a sense, both projects seem to converge towards the same goals, but from different directions. It should again be noted that World Foundry, like the source code in this book, is an open source (GPL) project and can use your help as a 3D programmer.

What Does This Mean for 3D Programmers?

While 3D content creation systems represent a fascinating and exciting way of creating interactive 3D applications, it is important to realize that the paradigms for interactive 3D content creation are still fairly young. Therefore, these packages show one possible direction for 3D development to take, but are not the ultimate answer to 3D.

In the end, it is the 3D programmers who create these systems in the first place, and who can extend these systems to introduce new features or even to establish new paradigms and ways of thinking about 3D content creation. A 3D content creation tool can help you realize your ideas

more quickly, but 3D programming knowledge allows you to use these systems to their full extent. So, don't think that the presence of these development kits obviates the need for 3D programming. No off-the-shelf (or off-the-Internet) program can be everything to everyone. It follows that there is always room for improvement in any 3D program, even in polished 3D content creation systems.

As a 3D programmer, you can make such improvements. If you learn about a new and interesting 3D rendering or visibility technique, then by all means try to implement it. Try to integrate it into your favorite 3D program. Discuss the idea with the author of a 3D content creation system, and see if it is possible to incorporate it. Consider the content creation tools as just that—tools. Good programmers understand the tools they use. They are able, willing, and eager to look under the hood, understand the program structure and concepts, and make improvements.

Without third-party tools, you would have to program all of your 3D tools yourself, which takes time. On the other hand, without 3D programming knowledge, you would be at the mercy of your tools; a program bug or lack of an important feature could be crippling for a project depending on that feature. Only the combination of good tools and a solid foundation in 3D programming equip you to tackle any problem and any project in the field of 3D graphics.

The Future

Chaos theory tells us that, in the long run, we can never predict anything about the future with any accuracy. Nevertheless, that never stopped anyone from trying, myself included. What follows are some speculations about possible future trends in the field of interactive 3D graphics, under Linux and in general.

Physically based simulations have already started to increase in number, and this trend will surely continue. There are already plug-in physics libraries that interface with an existing 3D library to add physics capabilities to the engine. Good simulation of rigid body dynamics will probably remain the focus for some time to come, but even items such as cloth are being animated in real time.

The field of VSD remains wide open for research. Scenes can always become more complex, meaning that efficient ways of culling large invisible parts of the scene are vital. The use of billboard-like techniques (collectively called *image-based rendering*) and dynamic LOD schemes can offer good visual quality while greatly reducing the work that needs to be done.

Simulating humans is always a challenging goal, one which everyone seems to strive for. Facial animation, character animation, inverse kinematics, and physically based modeling of humans are all topics which have received and will receive much attention. Human-like behavior, in other words AI, will also remain important.

Non-polygonal representations of data have been on the rise. Curved surfaces, voxels, and subdivision surfaces are some of the techniques being used today.

Content creation tools have started to change the way that 3D programs are created. There is still no universal paradigm that can be applied, but the tools do provide interesting approaches.

Finally, people will always want larger, more detailed virtual worlds. Developing algorithms to deal with such large data sets is challenging. Also, there is the question of how to generate such

large worlds in the first place. Some pseudorandom techniques can be used to deterministically generate huge amounts of data automatically, such as landscapes. Data for real-world locations can also be generated automatically from satellite and photographic information. As the demand for larger worlds grows, so will the need for such automated algorithms. Networking large virtual worlds is also an interesting issue. There is still no universally accepted standard for transmitting 3D content over the WWW, nor for networking various participants into collective virtual worlds. We can expect to see much work done in the network sector, since networking has the potential to make interactive 3D graphics environments available to practically every computer user.

Summary

In this chapter, we took a quick look at two 3D content development systems currently available for Linux. We saw how to create worlds in Game Blender and in World Foundry. We noted that World Foundry is an open source project, released under the GPL, which could use your skills as a 3D programmer. We finally speculated about the future directions 3D graphics programming may take.

Perspective

It is only appropriate that a book on 3D graphics ends with a section entitled "Perspective." Indeed, perspective is the one common thread running through all areas of 3D graphics. It is the compelling visual cue which creates the illusion of depth; it is the single most important formula in 3D graphics; it is the visual and geometric phenomenon which causes the non-linearity of z in screen space. Now, let's try to put the contents of this book into perspective, in the broader field of 3D graphics as a whole.

We can broadly classify 3D graphics into a spectrum with three categories: high-end graphics, real-time graphics, and network graphics. This book focused on real-time 3D graphics under Linux; we looked at the techniques and tools needed to produce interactive 3D programs with acceptable visual quality and interactive frame rates. This is the middle of the spectrum.

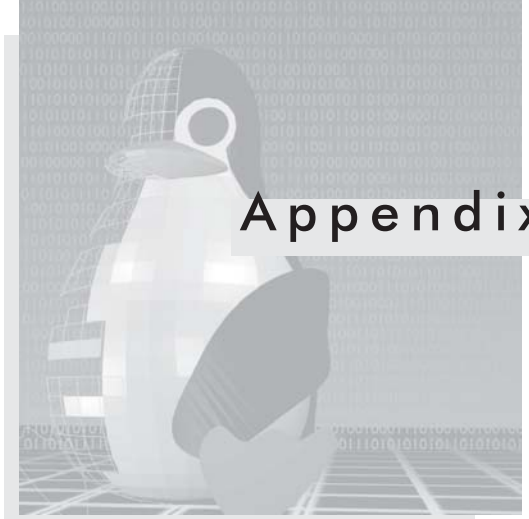
High-end graphics is at the high end of the spectrum. It typically involves algorithms which produce extremely high-quality photorealistic or hyper-realistic images, but which require enormous amounts of time and space to create. As processors get faster and as algorithms improve, techniques trickle down from the high-end scene into the real-time scene.

At the low end of the spectrum is network graphics. 3D graphics designed to be viewed from or manipulated through a WWW browser are typically much simpler than the models for real-time and high-end graphics, for two reasons. First, the transmission time for the data over the network must be minimized, leading to smaller and simpler models. Second, the target computer on which the network graphics are ultimately displayed is often not as powerful as a graphics workstation. In a network environment, we can make no assumptions about the computing power of the viewer's computer. Therefore, network graphics models are often optimized for the lowest-end machine.

We therefore can see an interesting progression of technology from the high-end scene, into the real-time scene, and finally into the network scene. Compare the visual quality of a ray-traced image exhibiting radiosity and light diffraction with the visual quality of a typical 3D game. Then, compare the visual quality of a 3D game with that of a VRML display in a WWW browser. The technological status of each arena becomes clear. Each area has its own constraints and its own goals, but when hardware or software technology improves, all three ranges of the spectrum benefit simultaneously, causing a migration of technology from the higher to the lower levels.

Thus, the contents of this book fit into the middle of the 3D graphics spectrum. Look at techniques from the high end to understand what might affect real-time 3D graphics in the future. Try to implement current techniques on the low end to make 3D graphics available to the net at large.

Having put the contents of this book into perspective, I can now in good conscience bring this chapter and the book to a close. I sincerely hope you have enjoyed our journey together through the exciting territory of Linux 3D graphics programming. From here, you are on your own. Hopefully, the material and code in this book have provided you with enough information and inspiration to create your own exciting 3D programs. 3D graphics is a very exciting, very dynamic, and very rewarding area. I encourage you to continue your exploration of the field, and also to share your work with the Linux 3D graphics community at large. Check out this book's web site at `http://www.linux3dgraphicsprogramming.org`, where you can find more information, leave feedback, submit and download improvements to the l3d library, or join an open source project. Remember, your own ideas, algorithms, and programs all can make a difference. You can help shape the future of Linux 3D graphics.



Appendix

CD Installation

License

The source code in this book was written by myself, Norman Lin, and is Copyright 2000 Norman Lin. This version of the source code has been released under the GNU General Public License (GPL). Please read the file `l3d/COPYING` on the CD-ROM. The other software packages on the CD-ROM (such as Blender, RPlay, Calc, and so forth), located in directory `software`, were written by other individuals or companies and may be subject to other licenses; read the documentation for each software package carefully.

Essentially, the GPL allows you to use this software in your own projects, but also requires you to make the source code to such projects available under the GPL. The GPL therefore encourages the sharing of information and source code among software developers. The GNU/Linux operating system was written in this spirit; developing the code in this book would not have been possible without the availability of free, open source, high-quality software and tools. Releasing my code under the GPL is my way of giving something back to the community. Check out this book's web site, `http://www.linux3dgraphicsprogramming.org`, to find the latest source code and related information. You can also download or submit your own source code improvements to l3d (via a common CVS repository), leave your comments, discuss the material with others, or join an open source project. I encourage you to do all of these things.

Note that the GPL permits you to sell your programs for a fee, even those programs that are based off of l3d or other GPL source code. The "free" in free software refers to freedom, not price. The purpose of requiring you to provide source code is not to try to prevent you from earning money based on the sale of your software; the purpose is to ensure that users of your program receive the same freedoms that you receive when you choose to use GPL source code as a basis for your own code. You must therefore make the source code to your software available so that others have the freedom to analyze, modify, improve, and use your software, just as you have the freedom to analyze, modify, improve, or use the software in this book.

Contents of the CD-ROM

Here is a brief list of some of the more important software and files included on the CD-ROM. There is, in reality, much more software on the CD-ROM (since the Linux operating system and

tools are included), but here the focus is on the software and files relevant to 3D graphics programming.

- l3d library and sample programs
- Color versions of all the figures from the book
- Animated videos of 3D concepts to supplement the static 2D figures, in AVI format
- Xanim: Video file player to play the AVI files
- Blender 1.80: Complete 3D modeling and animation suite
- World Foundry: 3D game development kit designed to enable non-programmers to construct complete 3D games
- Cygnus gcc: C++ compiler and tools for Microsoft Windows (can be used to port l3d to Windows)
- GIMP: GNU Image Manipulation Program, for editing 2D images
- Mesa: OpenGL compatible library
- GLUT: Mark Kilgard's GL Utility Toolkit
- Calc 2.02f: Symbolic algebra system
- RPlay: Digital sound server
- XFree86 4.0: Free X Window System with support for 3D hardware acceleration in a window

For those software packages or libraries distributed under the GPL, you can obtain the source code by following the links on this book's web site, `http://www.linux3dgraphicsprogramming.org`.

TEAMFLY

Quick Start Guide

This section describes the essential information to install the software on the CD-ROM. It reviews the locations of files on the CD-ROM and basic installation instructions, including the order in which the various software packages should be installed.

This section assumes you already have a working Linux installation with the X Window System (XFree86) installed. If this is not the case, first install Linux and then the X Window System. Exactly how to do this depends on your particular Linux distribution; see your system documentation for details.

Directories

The directories on the CD-ROM are organized as follows:

- `l3d`: Contains all of the example programs in the book and the l3d library, in `.tar.gz` format.
- `software`: Contains all of the third-party software and extra libraries used in the book, including Blender, GIMP, Mesa, RPlay, Calc, and XFree86 4.0.
- `figures`: Contains color versions of all of the figures and equations in the book, in PCX format.

Team-Fly®

- `videos`: Contains animated 3D videos illustrating 3D concepts, to supplement the flat 2D pictures in the book.

Installing the Sample Programs and Other Software

1. You must already have a Linux system up and running, with the following software already installed: gcc 2.95, an X Server (e.g., XFree86 3.3.5), Mesa, and GLUT. All modern Linux distributions include these libraries. If your Linux distribution does not include some of these packages, you must find them from another source (e.g., download from the Internet) and install them manually. The following software is also useful: GIMP, the Gnu Image Manipulation Program, for editing 2D images such as texture maps; and the PPM utilities for converting PPM files to other formats from the command line.
2. Mount the CD-ROM to make it accessible: **mount /cdrom**.
3. Type **export LD_LIBRARY_PATH=/usr/local/lib**. This allows the supplementary libraries, installed into this directory in the next step, to be located by other programs during the installation process.
4. Install each of the following supplementary libraries in the following order: `glib-1.2.8.tar.gz`, `gtk+-1.2.8.tar.gz`, `libsigc++-1.0.1.tar.gz`, `gtkmm-1.2.1.tar.gz`, and `rplay-3.2.0b5.tgz`. All of these libraries are located in `/cdrom/software`. The first four are needed by the Blender attributes editor program of Chapter 6; the last library is the RPlay sound server described in Chapter 8.

4a. Make a temporary directory to contain the files during installation. For instance, to install the glib library, type **mkdir glib_work**. Change to the directory you just created: **cd glib_work**.

4b. Unpack the files into the current work directory. Type **tar xzvf /cdrom/software/LIB.tgz**, where LIB is the name of the library being installed (e.g., `glib-1.2.8.tar.gz`). This should create a new subdirectory, such as `glib-1.2.8`. Change to this directory: type **cd glib-1.2.8**.

4c. If desired, read the documentation for the software package. Typically, this documentation will be distributed among a number of text files: the files `README` (describing the purpose of the software), `INSTALL` (installation instructions), and `COPYING` (software license agreement) are common. There may also be a subdirectory `doc/` in which additional documentation is located.

4d. Compile the library. Type **./configure; make**. (Though all of the libraries listed above require the configuration step with **configure**, not all software requires this step, in which case you would simply type **make**.) Compilation should complete without errors. If an error occurs, examine carefully the output of the command to determine the reason for failure, correct this, and try again. One possible reason for failure is the absence of other libraries required by the library being compiled. In this case, you must install the prerequisite libraries first.

4e. After successful compilation, install the library. Type **su** and press **Enter**, enter the root password, and press **Enter**. You are now the superuser. Type **make install** to install the compiled binaries into the proper locations on the system. Type **exit** to end the superuser session.

4f. Remove the temporary work directory. Assuming the temporary work directory was `glib_work`, type **cd ../..** to change to the original top-level directory from which you started the installation. Then, type **rm -rf glib_work** to remove the work directory.

4g. Repeat this process for each of the supplemental libraries to be installed.

5. Install the l3d sample programs as follows.

5a. Choose a directory where you want to install the sample programs, create it, and change to it. For instance, **mkdir l3d;cd l3d**.

5b. Set the `L3D` environment variable to point to the l3d directory, which is the current directory into which you are about to unpack the sample programs. Type **export L3D=`pwd`** and press **Enter**. Note that the quotation marks are single back-quotes, not normal quotes.

5c. Unpack the sample programs: **tar xzvf /cdrom/l3d/l3d-0.5.tar.gz**.



NOTE It is recommended that you recompile the sample programs as described in the steps below. However, the CD-ROM also contains precompiled binary versions of the programs.

5d. Edit the file `Makefile.vars` to set up the proper directory names for your system. The defaults work for the Debian GNU/Linux system, but may need to be changed for other Linux distributions. In particular, the variable `GLIDEDIR` should be set to the directory containing the Glide libraries, in case you are using Glide for 3DFX hardware acceleration. `MESAINCLUDEDIR` should point to the directory containing the Mesa include files. `MESALIBDIR` should point to the directory containing the Mesa binary library files `libGL.a`, `libGLU.a`, and `libglut.a`. The standard values should be correct for most systems.

5e. Type **make -f makeall.lnx** to build all of the sample programs. Remember, compiling the `blend_at` program requires that the following libraries be installed: `glib 1.2.8`, `gtk+ 1.2.8`, `libsigc++ 1.0.1`, and `gtkmm 1.2.1`.

6. Install Blender. Blender installation is different than installation of the other packages because Blender is distributed without source code in a precompiled binary form. Install Blender as follows.

6a. Make a directory to store the Blender files, and change to it. For instance, type **mkdir blender1.80; cd blender1.80**.

6b. Unpack the Blender executable files into the current directory. Type **tar zxvf /cdrom/software/blender/blender1.80-linux-glibc2.1.2-i386.tar.gz**. This creates a new subdirectory. Change to this subdirectory: type **cd blender1.80-linux-glibc2.1.2-i386**.

6c. Set the Blender environment variable to point to the current directory containing the Blender files. Type **export BLENDERDIR=`pwd`**. Note the back-quotes.

6d. To start Blender, type **./blender** in the Blender directory.

To install any other programs and libraries in the directory `software` unpack each `.tar.gz` file into a separate directory, and read the installation instructions for each package.

Troubleshooting the Sample Programs

If you have problems running the sample programs, here are some tips for finding the problem.

- Ensure that the `DISPLAY` environment variable is set correctly and that you can start other X programs from the command line.
- Ensure that all required libraries (Mesa, GLUT, RPlay, glib, GTK) have been installed.
- Ensure that any data files needed by the program are accessible or in the current directory. Such files include texture image files, texture coordinate files, object mesh files, world data files, and plug-in shared library files.
- Ensure that world data files and mesh files contain no invalid polygons. An example of an invalid polygon would be one with only two vertices, since a valid polygon must contain at least three vertices.
- If all else fails, try compiling the sample code with debugging. Edit the file `Makefile.vars`, uncomment the line containing "DEBUG=-g," and comment out all other lines referencing "DEBUG." (In Makefiles, the comment character is the pound sign #.) Then, type **cd $L3D;make clean;make -f makeall.lnx** to recompile all programs with debugging. Finally, execute the program in question with the debugger to determine exactly where the program crashes and why.

Some Comments on the Sample Programs

This section lists some various observations about the l3d code and the sample programs in the book.

The source code on the CD-ROM has additional comments within the C++ source code which, for space reasons, were omitted in the listings in the book. This removal of comments was done via an automated Perl program, so you can be sure that the code contents of the listings in the book are identical to those on the CD-ROM. The only difference is that there are ample comments in the code on the CD-ROM.

As mentioned in the introduction, the structure of l3d emphasizes a bottom-up approach where all code is understandable within the context of one chapter. This leads to heavy use of virtual functions. This results in code which is slightly slower than it could be if all of the virtual function calls were eliminated. For instance, the virtual call to `fog_native` for each pixel within the inner loop of the *z* buffered rasterizer incurs quite a performance overhead. Also, the members of all list objects are accessed through abstract pointers. For instance, the `ivertex` lists within the polygons initially contain simple integer objects, but later get extended to contain integer index and texture coordinate information. To avoid newly declaring the lists (and rewriting all code using the lists) in derived polygon classes, the lists are essentially virtual; only the objects are swapped out. This is quite convenient in terms of programming, but is somewhat inefficient; in reality, we practically always store texture coordinates with the vertex indices, so there is no reason (other than pedagogical) to maintain both the simple and the textured `ivertex` class with the corresponding virtual references. Similarly, our 3D objects contain virtual lists of polygons, although the most commonly used polygons are textured, light mapped polygons. Eliminating

some of the virtual function calls and abstract references leads to less flexible but possibly faster code.

The Mesa 3D rasterizer class uses 2D clipping and a reverse projection to obtain the original 3D coordinates of the 2D points (needed for perspective correct texture mapping and *z* buffering). A more efficient approach is to do all the clipping in 3D, meaning that we always have the original 3D coordinate of every single vertex plotted in 2D (which is not the case for new vertices created during 2D as opposed to 3D clipping). The reason that l3d uses the 2D clipping and reverse projection strategy is, again, pedagogical: we covered 2D clipping first, then for texture mapping we needed the original 3D coordinates based on the 2D clipped coordinates. But, after realizing later that we can do all the clipping in 3D, we can actually do away with the 2D clipping entirely, which can lead to more efficient code because we don't need to do any more reverse projections.

Again, check this book's web site at `http://www.linux3dgraphicsprogramming.org` to see the latest version of the l3d code, which may incorporate some of the improvements above.

Hardware Acceleration

In general, l3d programs can all take advantage of hardware acceleration by using OpenGL as the rasterization layer. This means that OpenGL must be configured to use hardware acceleration in order for l3d programs to use hardware acceleration.

There are two ways for you to use hardware acceleration. The first way requires a 3DFX graphics card, the Glide library, and a Mesa distribution compiled with Glide support. This means that you need to manually install Glide and compile Mesa; see the Mesa documentation for more information. There are many versions of the Glide library, for different generations of 3DFX hardware. I have included two versions of Glide on the CD-ROM, which should work with Voodoo and Voodoo3 cards. If you have a different 3DFX card, check out the web site `http://linux.3dfx.com` to download the proper version of Glide for yourself. After installing Glide and compiling Mesa to use Glide, you must edit the file `$L3D/Makefile.vars` to point to the correct directories. Variable `GLIDEDIR` should point to the directory containing the Glide library; `MESAINCLUDEDIR` should point to the directory containing the include files used to compile Mesa; and `MESALIBDIR` should point to the directory containing the new, 3DFX-enabled Mesa libraries. Also, you must uncomment the line starting with "GLIDE_LIBS," to cause l3d programs to be linked with the Glide library. After following these steps, you can use 3DFX/Glide hardware acceleration.

The second way of using hardware acceleration involves using XFree86 4.0 and the Direct Rendering Infrastructure (DRI). This requires you to compile the XFree86 4.0 system (included on the CD-ROM) with support for your 3D graphics card, and for you to use the special DRI Mesa/OpenGL libraries, which use the DRI to communicate directly with the underlying graphics hardware. This is similar to using Mesa to access 3DFX hardware through Glide, but with broader support for various hardware and with the ability to display accelerated 3D graphics within a window, as opposed to full-screen. Using the DRI for hardware acceleration does not require you to change the file `Makefile.vars`. However, you do need to ensure that the DRI OpenGL libraries, which use hardware acceleration, are dynamically linked to your executable program. To test

this, type **ldd PROGRAM**, where PROGRAM is the name of the executable program which should use hardware acceleration. You should see a line indicating that the `libGL.so` file is dynamically linked with the `libGL.so` file from the Xfree86 4.0 directory. If some other `libGL.so` file is linked, then this other OpenGL library will most likely be unable to use the DRI hardware acceleration. To control which directories are searched during dynamic linking, set the environment variable `LD_LIBRARY_PATH`. See the man page on the dynamic linker `ld.so` for more information. The XFree86 4.0 documentation also provides information on troubleshooting the DRI.



TIP The Utah-GLX project contains drivers for some 3D cards which are not yet supported by the DRI. Eventually, the Utah-GLX drivers will be integrated into the DRI, but until this happens, you may still find it useful to use the Utah-GLX drivers directly. The home page for the Utah-GLX project is `http://utah-glx.sourceforge.net`.

Porting the Code to Microsoft Windows

The l3d code has been designed so that it can be easily ported to other operating systems, such as Microsoft Windows. With a few exceptions, the sample programs in this book compile and run under Windows. However, support for Windows has not been a primary concern for l3d, since it was written to illustrate graphics programming under Linux, and since I have neither installed nor need any Microsoft products whatsoever on my main computer. Under the (not necessarily true) assumption that Microsoft and Windows continue to maintain some short-term market relevance, it might be interesting for you to make your programs available under Microsoft Windows as well.

To compile the code under Microsoft Windows, you can use the Cygnus GCC compiler, included on the CD-ROM in directory `software/ms-win`. This is a free C++ compiler and debugger which runs under Microsoft Windows and which can be used to compile the l3d code. Be sure also to read and install the OpenGL header files and libraries included with the Cygnus GCC compiler. The OpenGL files are located in the archive file `opengl-1.2.1- 1.tar.gz` on the CD-ROM.

The file `makecyg.bat` is a batch file which compiles all of the examples for Microsoft Windows by using the Cygnus GCC compiler. The following are Windows-specific source code files, which you should use as a starting point for understanding the ported code:

- `fwin.cc`, `fwin.h`: factory classes for creating Microsoft Windows factories
- `sc_mswin.h`, `sc_mswin.cc`: screen classes for creating a 2D output window using a device-independent bitmap (DIB) section
- `dis_mswin.h`, `dis_mswin.cc`: dispatcher classes for event handling under Microsoft Windows
- `sc_wogl.h`, `sc_wogl.cc`, `sc_wzogl.h`, `sc_wzogl.cc`: screen classes for creating an output window using OpenGL under Windows

The binaries directory for Windows object and executable files is `$L3D/binaries/cygwin`. The porting was performed by duplicating the directory structure from the Linux directory, and by editing the Makefiles as necessary.

By default, the Windows code is compiled with both a software renderer and an OpenGL-based renderer. This means that you should have OpenGL and the GLUT libraries installed on your system. OpenGL libraries should come standard with the operating system, and are also distributed as specialized drivers with any modern 3D graphics card. The GLUT library (file `GLUT32.DLL`) is included in the file `opengl-1.2.1-1.tar.gz` included with the Cygnus GCC compiler on the CD-ROM. If your OpenGL driver supports hardware acceleration, then the l3d programs will also run with hardware acceleration under Microsoft Windows. In the unlikely event that you do not have any OpenGL libraries at all installed on your system, you can change the Makefiles to compile and link without the OpenGL-specific files; in other words, it is also possible to compile l3d using purely software rendering under Windows.

Plug-ins do not yet work with the Microsoft Windows version of the code. There appears to be some difficulty in using Linux-style plug-ins with Cygnus GCC. For this reason, any programs using plug-ins will not currently work under Windows. Currently, the code is relying on an emulation of the Linux `dlopen` system call to load the libraries. However, Windows has its own dynamic library loading mechanism, `LoadLibrary`, which could be used instead.



NOTE Using a scripting language for dynamic behavior instead of plug-ins would avoid the problems of system-specific dynamic library conventions, but with a possibly slower run-time performance.

Be sure to check this book's web site at `http://www.linux3dgraphicsprogramming.org` to see the status of the Microsoft Windows l3d code and to download the latest version. Alternatively, you can choose to abandon Microsoft Windows entirely, which for the long term may be a better solution.

Tools Used to Prepare this Book

At the suggestion of my technical editor, Kevin Seghetti, I'll now briefly go over the tools used to prepare this book.

This book was written using exclusively Linux tools. The Linux distribution was SuSE 6.3, though I am currently in the process of migrating to Debian, because of Debian's clear commitment to the free software community.

The text was prepared using Corel WordPerfect 8 for Linux. I would have preferred to have written the text in LyX, which is a front end to the LaTeX document preparation system often used in academic circles. However, at the time of this writing, there exists no reliable way of converting LyX/LaTeX files to the proprietary word processing format required by my publisher. Sure, there were several tools that could do most of the conversion (`latex2html`, for instance), but after trying around five different packages, I found no solution that preserved fonts, lists, indentation,

headers, and so forth. So I used WordPerfect, which could export files in exactly the format needed by the publisher.

The program listings were stored separately from the text, as subdocuments. A Perl script stripped extraneous comments from all source code files and converted the ASCII source code files into RTF files. The RTF files were then linked into the master document. The code snippets within the text were delimited by specially formatted comments in the code; another Perl script then extracted these regions, and again created RTF files which could be bound into the master document. The scripts are in the l3d distribution, as files `make_clean.sh`, `no_double_space.pl`, and `snippets.pl`. Whenever I needed to update the source code, I would then merely run my scripts again, and the updated source code was automatically rebound into the master document.

The 2D figures were generated using Xfig, a very flexible object-based (as opposed to bitmap) drawing program. The class diagrams were generated manually with TCM, the Toolkit for Conceptual Modeling. Other promising class diagram tools include Argo/UML (which I did not use because it focuses on Java), and doxygen (which automatically produces excellent documentation from uncommented C++ code, and which I will probably use in the future). The 3D figures were generated with Blender. The animations on the CD-ROM were also created with Blender. Post-processing work on the figures was done with GIMP.

The equations were generated using LyX, an almost WYSIWYG environment for LaTeX. I created one LyX file containing all equations for a particular chapter. Next, I exported the LyX file as a LaTeX file. Then, I used latex2html to convert the LaTeX code into HTML; latex2html has the wonderful feature that it automatically converts LaTeX equations into bitmap image files. These bitmap image files were then renamed into the `eqnxx-xx` files on the CD-ROM.

To manage the large number of figures and equations for each chapter, I again wrote a Perl script which read an input file containing a list of all the figures for the chapter, in order. These figures were then converted into the format needed by the publisher and assigned sequential filenames. This way, I could work on the figures separately, with different file formats, and not worry about their exact order. If a figure got added or deleted, I simply made the appropriate change to the list of figures, and all the figures got assigned the correct numbers during the next run of the script. The PBM utilities were used within the script file to convert the varying image file formats into one common format.

Resources

The following is a list of some printed and online resources which may be of use to you as you continue to explore—and hopefully, advance the state of—Linux 3D graphics. The next section is a comprehensive list of the specific literature referenced in this book, but this section aims to present you with some more generally useful resources.

3D Graphics Programming

Computer Graphics, Principles and Practice by James Foley, Andries van Dam, Steven Feiner, and John Hughes. Regarded by many as a classic in the field, this book covers an extremely broad range of topics relevant to 2D and 3D graphics programmers. Practically every important topic is covered, even if briefly. This book has a somewhat mathematical slant to it.

OpenGL Programming Guide by Mason Woo, Jackie Neider, and Tom Davis. This is the definitive guide to programming with OpenGL.

Real-Time Rendering, by Tomas Möller and Eric Haines. This book contains numerous algorithms and concepts required for rendering high-quality graphics in real time. It is currently (March, 2001) one of the more up-to-date books in the field, and is a good collection of the most useful contemporary techniques.

`http://www.graphicspapers.com`. This site contains a very large searchable database of academic papers dealing with 3D graphics.

`http://www.flipcode.com`. This site offers many tutorials and columns relevant to programming real-time 3D graphics and games.

3D Modeling

Blender v1.5 Manual by Ton Roosendaal. This is the definitive guide to using Blender. See also `http://www.blender.nl`.

Das Blender Buch by Carsten Wartmann. This book provides a practical introduction and guide to using Blender, with several hands-on tutorials. (In German.)

Digital Character Animation 2, Volume 1: Essential Techniques by George Maestri. This platform-independent book illustrates the fundamentals of character animation with plenty of exercises and illustrations. Character animation is not easy, but this book helps you understand and master the difficulties.

Looking Good in 3D by Andrew Reese. This practice-oriented book explains basic principles of 3D imagery and illustrates which 3D graphics techniques to use in order to create effective presentations.

Drawing on the Right Side of the Brain by Betty Edwards. This book deals with developing the visual skills to become a good 2D artist. While it doesn't directly deal with computer graphics, the visual skills you can learn from this book are very effective for all types of visual expression, including 3D modeling.

Keys to Drawing by Bert Dodson. This book, too, does not deal directly with computer graphics, but offers a number of useful tips or "keys" to creating effective 2D art.

3D Information and Applications

`http://www.linux3dgraphicsprogramming.org`. This is the home page for the book, and provides links to many other resources, as well as discussion forums, hourly updated Linux 3D news, tutorials, articles, code repositories, and more. Check it out!

`http://www.worldfoundry.org`. This is the home page for the World Foundry system described in Chapter 9, a game development kit allowing creation of sophisticated, multi-platform 3D games with no programming.

`http://www.blender.nl`. This is the official home page of the Blender 3D rendering, animation, and game development suite.

`http://www.opengl.org`. The OpenGL home page provides news, links, and sample code dealing with OpenGL.

`http://mesa3d.sourceforge.net`. The Mesa home page provides news and the latest version of the Mesa library, which is effectively compatible with OpenGL.

`http://sunsite.auc.dk/linuxgames`. The Linux Games Development Center provides articles and news for developers of games under Linux.

General Programming

The X Window System: Programming and Applications with Xt by Douglas Young and John Pew. This book provides an introduction to programming the X Window System.

The C++ Programming Language by Bjarne Stroustrup. This is, of course, the definitive reference on the C++ language.

Object-Oriented Software Construction by Bertrand Meyer. This book is a comprehensive treatment of the subject of object orientation and how it improves software quality. The book is very readable, yet also provides rigorous treatment of what object orientation actually is, and its relationship to other software methodologies and to the field of scientific study as a whole. While much literature attempts to make object orientation appear easy, this book does not shy away from showing the true, subtle, real-world difficulties of software construction, and how object orientation attacks these problems head-on.

Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. This landmark book documented for the first time the concept of a design pattern, presented a catalog of reusable patterns, and encouraged the use of this technique for object-oriented development.

Data Structures and Program Design by Robert Kruse. This introduction to data structures describes many of the fundamental structures of computer science, which are essential in programs of all kinds, including 3D programs.

Other

University Physics by Francis W. Sears, Mark W. Zemansky, and Hugh D. Young. This undergraduate-level text provides a useful introduction to the field of physics, some of which can be applied to computer simulations.

The Guitar Handbook by Ralph Denyer. I have found that good programmers are often also good artists—musically, visually, or otherwise. If you write interactive 3D programs such as games, you might also need to write music to accompany these programs. This book, while focusing on the guitar, provides a good treatment of music theory as well, which is useful for composing music of all kinds.

`http://www.debian.org`. This is the home page for the Debian GNU/Linux distribution of the Linux operating system.

`http://www.gnu.org`. This is the home page of the GNU project, which is a pivotal proponent of free software (free as in freedom), and whose software forms a large and important part of the GNU/Linux operating system. The GNU project is the source of the GNU General Public License, or GPL, under which much Linux software has been released.

TEAM-FLY

References

CHRI96 Christianson, David B., Sean E. Anderson, Li-Wei He, Daniel S. Weld, Michael F. Cohen, and David H. Salesin. "Declarative Camera Control for Automatic Cinematography." *Proceedings of AAAI '96* (Portland, OR), pp. 148-155, 1996. `http://www.cs.washington.edu/research/grail/pub/ abstracts.html`.

DUCH97 Duchaineau, M., M. Wolinski, D. Sigeti, M. Miller, C. Aldrich, and M. Mineev-Weinstein. "ROAMing Terrain: Real-time Optimally Adapting Meshes." Online at `http://www.llnl.gov/graphics/ROAM`.

FOLE92 Foley, James, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice*. New York: Addison-Wesley, 1992.

GAMM95 Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading (Massachusetts): Addison-Wesley, 1995.

GARL98 Garland, Michael, and Paul S. Heckbert. "Simplifying Surfaces with Color and Texture using Quadric Error Metrics." *IEEE Visualization 98*, pp. 263-269, July 1998. `http://www.cs.cmu.edu/~garland/`.

GOLD98 Gold, Michael, Mark Kilgard, Richard Wright Jr., Jon Leech, and Matthew Papakipos. *Advanced OpenGL Game Development*. Course notes from the Computer Game Developers Conference, Long Beach, California, May 4-8 1998. `http://www.berkelium.com/OpenGL/cgdc98`.

Team-Fly®

GORD91 Gordon, Dan, and Shuhong Chen. "Front-to-back display of BSP trees." *IEEE Computer Graphics and Applications*, vol. 11, no. 5, pp. 79-85, September 1991.

HECK97 Hecker, Chris. "Physics." A four-part series of articles for *Game Developer*, 1995-1997. `http://www.d6.com`.

HOPP96 Hoppe, Hughes. "Progressive Meshes." *Computer Graphics (SIGGRAPH 96 Proceedings)*, pp. 99-108, August 1996. `http://research.microsoft.com/~hoppe/`.

LUNS98 Lunstad, Andrew. "How Bout Dem Bones?: A Guide to Building a Bones-Based Animation System," *Proceedings of the Computer Game Developers Conference 1998* (Long Beach, California), pp. 403-411, Miller Freeman Inc., May 1998.

MELA98 Melax, Stan. "A Simple, Fast, and Effective Polygon Reduction Algorithm." *Game Developer*, vol. 5, no. 11, pp. 44-49, November 1998. `http://www.cs.ualberta.ca/~melax/polychop/`.

MEYE97 Meyer, Bertrand. *Object-Oriented Software Construction*. New Jersey: Prentice Hall PTR, 1997.

MOEL99 Möller, Tomas, and Eric Haines. *Real-Time Rendering*. A K Peters, 1999.

NAYL98 Naylor, Bruce. "A Tutorial on Binary Space Partitioning Trees." in *Proceedings of the Computer Game Developers Conference 1998* (Long Beach, California), pp. 433-457, Miller Freeman Inc., May 1998.

PEIT92 Peitgen, Heinz-Otto, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals: New Frontiers of Science*. New York: Springer-Verlag, 1992.

RICH91 Rich, Elaine, and Kevin Knight. *Artificial Intelligence, Second Edition*. McGraw-Hill, 1991.

ROOS99 Roosendaal, Ton. Personal communication, May 1999.

SCHA96 Schaufler, G., and W. Strzlinger. "A Three Dimensional Image Cache for Virtual Reality," in *Proceedings of Eurographics 96,* pp. 227-236, 1996. `http://www.gup.uni-linz.ac.at:8001/staff/schaufler/papers/`.

SCHU69 Schumacker, R., B. Brand, M. Gilliland, and W. Sharp. *Study for Applying Computer-Generated Images to Visual Simulation*. Technical Report AFHRL-TR-69-14, NTIS AD700375, U.S. Air Force Human Resources Lab., Air Force Systems Command, Brooks AFB, TX, September 1969.

SEGH97 Seghetti, Kevin, et al. "World Foundry Reference Manual." Online at `http://www.worldfoundry.org`.

SEGH00 Seghetti, Kevin. Personal communication, September 2000.

SHAD96 Shade, J., D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. "Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments." *Computer Graphics (SIGGRAPH 96 Proceedings)*, pp. 75-82, August 1996. `http://www.cs.washington.edu/research/grail/pub/abstracts.html#HierImageCache`.

SHAN98 Shantz, Michael, and Alexander Reshetov. "Physical Modeling for Games." *Proceedings of the Computer Game Developers Conference 1998* (Long Beach, California), pp. 685-738, Miller Freeman Inc., May 1998.

SHAR99a Sharp, Brian. "Implementing Curved Surface Geometry." *Game Developer*, vol. 6, no. 6, pp. 42-53, June 1999.

SHAR99b Sharp, Brian. "Optimizing Curved Surface Geometry." *Game Developer*, vol. 6, no. 7, pp. 40-48, July 1999.

SHAR00 Sharp, Brian. "Optimizing Curved Surface Geometry." *Game Developer*, vol. 7, no. 1, pp. 34-42, January 2000.

SUTH74 Sutherland, I.E., and G.W. Hodgman. "Reentrant Polygon Clipping." *Communications of the ACM* 17(1), pp. 32-42, January 1974.

WILL83 Williams, Lance. "Pyramidial Parametrics." *Computer Graphics*, vol. 7, no. 3, pp. 1-11, July 1983.

WOO97 Woo, Mason, Jackie Neider, and Tom Davis. *OpenGL Programming Guide, Second Edition*. Reading (Massachusetts): Addison-Wesley, 1997.

Index

An asterisk indicates the reference includes code segments.

/dev/audio file, 517
2D graphics, 1-2
2D lumel space, converting to 3D camera space, 159
3D camera space, converting from 2D lumel space, 159
3D content development system, 583, 598-599
3D graphics, 3-4
3D graphics coordinate system, 4-5
3D morphing, 39-40, 41-47*
3D polygons, associating textures to, 96-98
3D rasterizer, abstract, 98-101*
3D rasterizer implementation,
 Mesa/OpenGL, 115-129*
 software, 101-113*
3D vector, 6
3DWindow, 33

acceleration, 578
Actor attributes, 439-440
actors, creating room with, 463-473
AI, 580-582
alpha channel, 485
ambient light, 50
animation tutorial, 180-201*
anti-aliasing, 345
arbitrarily oriented bounding box, 231-232
arm Ikas, creating, 184
artificial intelligence, *see* AI
attribute system, 431-433
attributes,
 associating with meshes, 431-433
 mesh, 437-440
 parsing, 440-446*
 used with VidscParser.pm, 437-440
 working with, 473-474
axially aligned bounding box, 232
axis-aligned billboards, 484-485
axis-aligned BSP tree, 302-303

back-face culling, 207
 example program, 209-214*
 of portals, 314
back-facing polygons, 207-208
 determining, 208-209
base, 181
billboards, 484-486, 506
binary space partitioning tree, *see* BSP tree
binary temporal search method, 542
binutils, 32
blend_at tool, 433-434, 446
 configuring, 434-436
 testing, 436
 using to create room with actors, 463-473
Blender,
 as a world editing system, 364-367
 interface, 33
 keystroke commands, 35
 steps for creating worlds, 474-475
 using to align portals, 374-382
 using to create portals, 371-374
 windows, 33-34
Blender attributes editor, *see* blend_at tool
Blender models, exporting to Videoscape format, 36
Blender tutorials
 creating 3D morph targets, 161-180*
 modeling walking human figure, 180-201*
Blender 2.0, using, 583-590
blender_config class, 447-448*
blender_controller class, 448-449*
blender_xcontroller class, 449-463*
body Ika, creating, 185
bone, 181
bounding sphere, 225
 checking, 225-227
 computing, 227-228
bounding volume, 225, 231-232, 531-532
 and BSP trees, 275-276
BSP tree, 268-270 *see also* axis-aligned BSP tree, leafy BSP tree, mini BSP tree
 and bounding volumes, 275-276
 and dimensions, 306
 example program, 276-293*
 level of detail techniques, 507-508
 representing volume with, 304-306
 traversing, 274-275
 using for world editing, 476
 using to pre-sort polygons, 271-274
 using to represent polyhedral solid, 303-306
bump map, 499
bump mapping, 499-500
ButtonsWindow, 33

Calc tool, 79
 commands, 81-82
 features, 83-85
 using, 79-82
 using to solve systems of equations, 85-86

using to solve texture mapping equations, 86-89
camera coordinates, defining frustum in, 218-220
camera position and portals, 315-316
camera tracking, 512-513
CD-ROM, installing, 603-609
cell, 308
chain, 181
character animation, 180
character special device, 517
characters, 180
child process, 528
client, IP, 524-525
clipping, 25-26
portals, 314-315
clouds, simulating, 497-498
collision detection, 530-531
testing for intersection, 532-538
collision detection example program, 574-576*
optimizing, 576-577
collision response, 531, 541
collisions,
multiple, 541, 542-543
types of, 536
column vector notation, 6
common palette, generating, 73
compatible morph targets, 161
configuration space, 581
control net, 503
controlling script, 429-430*
convexity, 209, 295
of portals, 315
coordinate system in 3D graphics, 4-5
correctness, VSD, 204
cubic environment mapping, 482-483
curved surfaces, 501-504
cyclic overlap, 243-244

diffuse reflection, 50-54
diffuse surfaces, 50
digital sound, *see* sound
dimensions and BSP trees, 306
dispatcher, 17
creating, 18
starting, 18
display list, 125
double buffering, 2
dynamic lighting, 58-59
dynamics, 578

edge collapse, 506-507
effector, 181
efficiency, VSD, 204-205
Emacs editor, 32
environment map, 481
environment mapping, 480-484
Euler integration, 490
event dispatcher, 18
extrusion, creating portals via, 371-374

factories, choosing, 13-14
factory management, 24-25
factory manager, 13
for z buffering, 261-263*
FileWindow, 33
find, 32
fire effects, 498-499
fixed-point arithmetic, 29
FK, 181-182
programming, 200-201
flat shading, 61
float type, 29
floating-point arithmetic, 29
fog effects, 59-60, 499
forward kinematics, *see* FK
fractal landscapes, 510
free edge, 379-370
front-facing polygons, determining, 208-209
frustum, computing in world coordinates, 220-221

Game Blender, using, 583-590
gcc, 32
gdb, 32
Gimbal lock, 385
Glade, 446
Gouraud shading, 61-62
graphical techniques, 501-513
gravity, 488

half space, 268
splitting, 268-269
haptic devices, 516
hardware acceleration, 2
hashes, 387-388
height field, 509
hidden surface removal, *see* HSR
hierarchical view frustum culling, 223-225
hierarchy, creating for Ikas, 185-187
homogenous coordinates, 6
HSR, 205
hyperplanes, 306

IK, 181-182
programming, 200-201
IK chain, 181
Ika, 181
associating with mesh, 189-190
parenting into hierarchy, 185-187
Ika chains, 181
animating, 188-189
creating, 183, 184-185
moving, 181-182, 183
Ika meshes, importing, 192
ImageSelectWindow, 34
ImageWindow, 33
info command, 32

InfoWindow, 33
intersection testing, 532-538
inverse, *see* matrix inverse
inverse kinematics, *see* IK
IP address, 526-527
IP client, 524-525
IP server, 526-528
IpoWindow, 33

joints, 181

l3d directory structure, 12
l3d example program, 8-11*
l3d program structure, 13-19
l3d_bounding_sphere class, 228-231*
l3d_bsptree class, 277-285*
 leafy BSP tree methods, 296-297
l3d_camera_collidable class, 552-553*
l3d_collidable class, 543-544*
l3d_collidable_sphere class, 544-548*
l3d_coordinate class, 25
l3d_dispatcher class, 20
l3d_event_source class, 19
l3d_factory_manager class, 24
l3d_factory_manager_v_0_3 class, 261-263*
l3d_halfspace class, 277-278*, 285
l3d_list class, 25
 using, 26
l3d_matrix class, 28
l3d_object class, 28
l3d_pipeline class, 15, 19
l3d_pipeline_world class, 29
l3d_pipeline_world_lightmapped class, 351-353*
l3d_plugin_videoscape_mesh_seeker class, 563-574*
l3d_point class, 25
l3d_polygon_2d class, 28
l3d_polygon_3d class, 28
l3d_polygon_3d_collidable class, 548-551*
l3d_polygon_3d_flatshaded class, 28
l3d_polygon_3d_portal class, 322-323*
l3d_polygon_3d_textured class, 96-97
l3d_polygon_3d_textured_lightmapped class, 142-145*
l3d_polygon_3d_textured_lightmapped_collidable class, 551-552*
l3d_polygon_ivertex class, 96
l3d_polygon_ivertex_textured class, 96
l3d_polygon_ivertex_textured_items_factory class, 96
l3d_rasterizer_2d class, 23
l3d_rasterizer_2d_imp class, 23
l3d_rasterizer_2d_mesa_imp class, 23
l3d_rasterizer_2d_sw_imp class, 23
l3d_rasterizer_2d_sw_lighter_imp class, 344-351*
l3d_rasterizer_3d class, 98-100*
l3d_rasterizer_3d_imp class, 98-100*
l3d_rasterizer_3d_mesa_imp class, 115-124*
l3d_rasterizer_3d_sw_imp class, 101-113*
l3d_rasterizer_3d_zbuf_mesa_imp class, 258-261*
l3d_rasterizer_3d_zbuf_sw_imp class, 248-257*
l3d_real type, 29
 using, 30
l3d_screen class, 20-21
 merging with l3d_screen_info class, 21
l3d_screen_info class, 21-22
 merging with l3d_screen class, 21
l3d_screen_info_indexed class, 22
l3d_screen_info_rgb class, 22
l3d_screen_zbuf_mesa class, 258-261*
l3d_sector class, 323-329*
l3d_sound_client class, 521-522*
l3d_sound_server_rplay class, 522-524*
l3d_surface class, 138-141*
l3d_surface_cache class, 141, 142-145*
l3d_texture class, 66-68*
l3d_texture_computer class, 67-68*
l3d_texture_data class, 66-68
l3d_texture_loader class, 68-69*
l3d_texture_loader_ppm class, 69-73*
l3d_texture_space class, 66-68*
l3d_two_part_list class, 25
 using, 26-27
l3d_vector class, 28
l3d_viewing_frustum class, 221-222*
l3d_world class, 29
l3d_world_backface class, 214-218*
l3d_world_bsptree class, 286-290*
l3d_world_frustum class, 236-242*
l3d_world_portal_textured_lightmapped_obj class, 329-344*
l3d_world_portal_textured_lightmapped_obj_colldet class, 553-563*
Lambert's law, 53
Lambertian surfaces, 53
landscapes,
 generating fractal, 510
 level of detail techniques, 511
 rendering, 511
 storing, 509-510
leafy BSP tree, 294-295, 304-305
 creating, 295-296
 example program, 297-302*
 methods for, 296-297
left-handed coordinate system, 4-5
leg Ikas, creating, 183-184
lens flare, 486-487
level editing, *see* world editing
level editor, *see* world editor
level of detail techniques, *see* LOD techniques
light intensity, 48
 computing, 319-320
light mapping, 64, 135-136
 and texture mapping example program, 151-159*
 implementing in software, 136-146
 implementing with hardware, 147-151

light maps, 63-64
and shadows, 159-160
rasterizing, 147
light rays, 3
light sources, computing multiple, 57
light table, *see* lighting table
light,
computing 49-59
rendering techniques, 60-64
shading techniques, 60-64
lighting models, 49-59
lighting table, 48
limb, 181
Linux
resources, 611-616
tools, 32, 610-611
LOD models, computing, 506-509
LOD techniques, 505-506
for landscapes, 511
lumel, 63

magic edges, 431, 433
magic vertices, 431
make utility, 32
man command, 32
material parameters, 50
matrices, 6
matrix
inverse, 7-8
multiplication, 6
properties, 7-8
transformations, 6-7
matte surfaces, 50
Mesa, 2
using to draw texture mapped polygons, 115-129*
z buffering in, 257-261*
Mesa/OpenGL 3D rasterizer implementation, 115-129*
mesh attributes, 437-440
meshes,
associating attributes with, 431-433
associating with Ika, 189-190
defining, 162-163
deforming, 165-167
exporting, 190-191
Microsoft Windows, porting code to, 609-610
mini BSP tree, 303
MIP map, 508
MIP mapping, 508-509
mirror, portal-based, 320
model, texturing, 191-192
morph targets,
exporting, 173
inserting, 163-165
morphing, 39
example program, 41-47*
tutorial, 161-180*

movie, loading, 198-199
moving collision, 536
multi-pass rendering, 500-501
multiple light sources, computing, 57
multiple sounds, playing, 518-519
multi-texturing, 501

natural phenomena, simulating, 497-499
near z clipping of portals, 316-318
network,
exchanging data on, 530
interface, 526
non-blocking operations, 529-530
non-convex sector-based partitioning for world editing, 476-477
non-splitting nodes, 304
numerical integration, 579-580

objects, specifying, 28-29
occlusion-based algorithms, 356-357
occlusion map, 356
octree, 306-308
using for world editing, 476
OopsWindow, 33
OpenGL, 2
z buffering in, 257-261*
orthonormalization, 564
overdraw, 243

painter's algorithm, 242-245, 274-275
parent process, 528
particle systems, 487-488
and physics, 488, 496
example program, 490-496*
particles, 487
patch, 501-502
path finding, 581
penetration,
allowing, 541-542
avoiding, 542-543
Perl generator module, 422
Perl modules, 388, 390
Actor.pm, 408-410*
errGen.pm, 426-429*
Facet.pm, 392-398*
l3dGen.pm, 422-426*
Portal.pm, 400*
Sector.pm, 400-408*
Texture.pm, 398-400*
Vertex.pm, 390-392*
VidscParser.pm, 415-422*
World.pm, 410-415*
Perl parsing module, 415
Perl portalization system, 389-390
executing, 429-430*
Perl scripts, using to convert Videoscape files, 387-389
Perl structural modules, 390
perspective correction, 126

perspective projection, 5-6
Phong illumination, 55
Phong shading, 63
physics,
 and particle systems, 488, 496
 basic concepts, 577-578
pipeline, 14
 creating, 18
pipeline subclass, declaring, 14-17
planar geometric perspective projection, 4
point light source, 50-51
points, 4
 specifying, 25
polygon soup, 290
polygons,
 clipping, 25-26
 pre-sorting, 271-274
 specifying, 28
port, 519
portable pixmap format, *see* PPM
portal algorithm components, 308-310
portal connectivity, 370-371
portal rendering and other rendering methods, 321-322
portal world example program, 353-355*
portal-based mirror, 320
portalization, 385
 in Perl, 389-390
portals, 309
 advantages and disadvantages, 313-314
 aligning, 374-382
 and camera position, 315-316
 and near z plane, 316-318
 back-face culling, 314
 clipping, 314-315
 creating, 371-374
 generating shadows with, 318-320
 rendering, 310-313
 working with, 382-385
PPM, 68
 utilities, 73-74
PPM files, converting, 73
principle of locality, 272
projection, 4
 perspective, 5-6

radial fog, 59-60
radiosity, 58
raw format, 518
ray tracing, 58, 357, 480-481
ray-to-polygon intersection testing, 532-535
ray-to-sphere intersection testing, 535-536
real-time update, 489, 579-580
reflection vector, 55
 computing, 55-57
rendering
 landscapes, 511
 light, 60-64
 multi-pass, 500-501
rendering methods, combining, 321-322
reverse projection, 127-128
right-handed coordinate system, 5
rigid body dynamics, 578-579
root, 181
rotoscoping, 197-200
RPlay server, 519
 using TCP/IP with, 520-521

sample programs
 backface, 209-214*
 bsp, 276-293*
 client, 525-526*
 collide, 574-576*
 drawdot, 8-11*
 frustum, 233-236*
 ika, 193-197*
 leafybsp, 297-302*
 lightmap, 151-159*
 morph3d, 40- 47*
 particle, 490-496*
 playtest, 520-521*
 porlotex, 353-355*
 server, 526-529*
 square, 518*
 textest, 129-135*
 texzbuf, 263-264
 vids3d, 175-179*
screen coordinates, converting to texture coordinates, 89-92
screen space coordinates, converting to world space coordinates, 127-128
screen-aligned billboards, 484-485
sector, 308
Sector attributes, 437-439
seen light rays, 3
self lighting, 49
SequenceWindow, 33
server, IP, 526-528
shading techniques, 60-64
shadows, 159-160
 generating with portals, 318-320
skeleton, 185
sky effects, 498
smoke effects, 498
smoothing groups, 62
socket, 520
sound, 516-519
 example program, 518*
sounds, playing multiple, 518-519
SoundWindow, 33
space partitioning, 268-271
span buffering, 264-266
spatial partitioning, 308

using for world editing, 475-476
special effects, 480-501
specular reflection, 55-57
sphere checking, 225-227
sphere map, 483
sphere mapping, 483
sphere-to-polygon intersection testing, 536-538
sphere-to-sphere intersection testing, 532
spherical environment mapping, 483
splitting nodes, 304
splitting plane, 268, 271-272
choosing, 272-274
spring force, 541-542
sprites, 484
square wave, 517
stability condition, 580
static collision, 536
static lighting, 58
strategy design pattern, 20
subdivision surfaces, 503-504
surface, 136-138
surface cache, 138, 141-142
sweep, 539
sweep test, 539-541

TCP/IP networking, 524
using with RPlay, 520-521
texels, 64
texture coordinates,
assigning in Blender, 167-173
converting screen coordinates to, 89-92
mapping to integer indices, 92-93
texture data, storing, 66-68*
texture definition triangle, 393
texture information, exporting, 174-175
texture mapping, 64-65
and light mapping example program, 151-159*
example program, 129-135*
pixel-level strategy, 89-93
strategy, 65, 94-95
vertex-level strategy, 93-96
texture mapping equations, solving with Calc, 86-89
texture objects, 125
texture space, 64
defining, 74-75
texture space and world space, mapping between, 76-89
textured room, creating with blend_at tool, 463-473
textures, 64
applying in Blender, 167-173
associating with 3D polygons, 96-98
defining, 65-74
loading, 68-73*
TextWindow, 33
tiles, 509-510
triangle strip, 502-503
tunneling, 538-541

two-part vertex lists, 26-27

updating in real time, 489

vector operations, 5
vectors, 4
velocity, 577
vertex class, 447*
vertex keys, 163
vertex lists, specifying, 25
vertices, specifying, 25
VFW axis, 41
Videoscape files, converting with Perl scripts, 387-389
Videoscape format, exporting Blender models to, 36
view frustum, 218
defining, 218-220
view frustum clipping, 233
view frustum culling, 218
example program, 233-236*
hierarchical, 223-225
view volume, 218
visual surface determination, *see* VSD
VRI axis, 41
VRML format, 174-175
VSD, 203
goals of, 204-207
VUP axis, 41

water, simulating, 499
world, 29
world coordinates, computing frustum in, 220-221
world design process, 367-368
world editing, 360
solutions, 360-363, 475-477
world editing system,
data flow, 368-369
requirements, 363-364
using Blender as, 364-367
world editor, 360
adapting, 362-363
custom, 361-362
world file, 36
generating, 385-386
world file format, 343-344
World Foundry, 590-591
using, 591-598
world partitioning techniques, 206-207
world space and texture space, mapping between, 76-89
world space coordinates, converting from screen space coordinates, 127-128

z buffer, 245
z buffer algorithm, 245-246
advantages and disadvantages, 246-247
z buffer rasterizer implementation,
Mesa/OpenGL, 257-261*
software, 248-257*
z buffering, factory managers for, 261-263*


Gamedev.net
The most comprehensive game development resource
The latest news in game development
The most active forums and chatrooms anywhere, with insights and tips from experienced game developers
Links to thousands of additional game development resources
Thorough book and product reviews
Over 1000 game development articles!
Game design
Graphics
DirectX
OpenGL
AI
Art
Music
Physics
Source Code
Sound
Assembly
And More!
Gamedev.net
OpenGL is a registered trademark of Silicon Graphics, Inc.
Microsoft, DirectX are registered trademarks of Microsoft Corp. in the United States and/or other countries.